ВСТУП

Метою даної дипломної роботи є розробка програмного забезпечення для розподіленої комп’ютерної системи зберігання даних. Система має відповідати наступним вимогам:

- можливість легкого горизонтального масштабування системи шляхом запуску програмного забезпечення на кожному новому вузлі із мінімальною зміною його налаштувань
- кожен вузол системи повинен бути здатним до паралельного обслуговування декількох клієнтських з’єднань
- система повинна надавати один інтерфейс для збереження, пошуку та отримання даних (Cloud-система)
- в якості програмної платформи необхідно використати сучасне, недороге (або безкоштовне) та надійне програмне забезпечення

Із розвитком та здешевленням обчислювальної техніки, сучасне суспільство все більше і більше залежить від неї — зараз практично не залишилося таких сфер діяльності суспільства, де б не використовувалися комп'ютерні технології.

Подібна залежність вимагає зростання обчислювальних потужностей для опрацювання величезних об'ємів даних, зростання об'єму інформації потребує все більшого об'єму накопичувачів для її збереження, підвищуються вимоги до надійності зберігання даних та їх доступності.

Забезпечення надійності роботи систем обробки та зберігання інформації вимагає значних фінансових вкладень: підтримання необхідних фізичних умов (температура, вологість) для апаратного забезпечення, надання безперебійного електропостачання, розробка і підтримка функціонування програмного забезпечення.

Цікавою особливістю нинішнього етапу розвитку ринку інформаційних технологій характеризується таким явищем, як “аутсорсінг” (англ. “outsourcing”) — передача організацією частини завдань стороннім виконавцям. Це дозволяє суттєво заощадити на утриманні штатних робітників.

Термін “аутсорсінг” можна застосувати не лише до працівників, але і до систем обробки даних: замість утримання власного центру обробки даних або потужних електронно-обчислювальних машин, їх можна орендувати на необхідний термін. Окрім того, подібна оренда може включати в себе не лише можливість використання апаратного забезпечення інформаційної системи, а й спеціалізованого програмного забезпечення.

Наявність попиту на обчислювальні ресурси значних потужностей стала поштовхом до розвитку “хмарних” (англ. “cloud”) обчислень — надання у використання ресурсів обчислювальної системи без необхідності користувача створювати та підтримувати її інфраструктуру і навіть розуміти процеси, які відбуваються всередині “хмари”.

Загалом, під “хмарою” розуміють обчислювальну систему, структура та принцип роботи якої неважливі для користувача.

“Хмарні” системи можна поділити за призначенням на дві основні групи:

- системи для високопродуктивних обчислень (англ. High-Performance Computing, HPC)
- системи для зберігання та обміну даними (англ. Content Distribution/Delivery Network, CDN)

Перша група орієнтована на швидке опрацювання великих об'ємів даних (наприклад, аналіз даних для прогнозування погоди), витримування великого навантаження (наприклад, веб-сайти із великою відвідуваністю) тощо.

Друга група (системи для зберігання та обміну даними) орієнтована не стільки на опрацювання даних, як на забезпечення надійності їх збереження, доступності в будь-який момент часу та швидкість доступу до них.

Основними вимогами до “хмари” є розподіл навантаження та можливість масштабування.

Масштабування може бути вертикальним (наприклад, заміна процесору на більш потужний) або горизонтальним (нарощування швидкодії системи за рахунок додавання нових вузлів у неї).

“Хмара” має відносно легко масштабуватися горизонтально, так як зростання, наприклад, тактової частоти процесорів зараз помітно призупинилося, тож розраховувати на підвищення продуктивності системи за рахунок використання найновішого апаратного забезпечення при зростанні навантаження не варто. Окрім того, часто горизонтальне масштабування дозволяє підвищити потужність обчислювальної системи набагато дешевше, ніж вертикальне. Подібна практика знайшла застосування у такій великій інформаційній компанії як Google.

Горизонтальне масштабування виконується за рахунок спеціально спроектованого програмного забезпечення. Тож при розробці потужної системи варто акцентувати увагу саме на ньому.

1 АНАЛІЗ ІСНУЮЧИХ МЕТОДІВ ТА ЗАСОБІВ ПОБУДОВИ CLOUD-СИСТЕМ

Найпростішими методами збільшення дискового простору серверу під керуванням операційної системи сімейства BSD/Linux за рахунок інших серверів є NFS (англ. Network File System — мережева файлова система), SSHFS (Secure Shell File System) та декотрі менш розповсюджені рішення.

Ці способи засновані на наступному принципі: директорії, доступні публічно або після авторизації, доступні між комп'ютерами у мережі. Віддалені диски сприймаються користувачем як директорії на сервері, із яким він працює в даний момент часу. Але передача даних (наприклад, при копіюванні або перегляді файлів) відбувається через мережу по зашифрованому каналу.

Для Windows-систем аналогічною є технологія Samba.

Недоліком даних способів є необхідність налаштовувати кожен новий вузол мережі для монтування дисків уже існуючих серверів.

Набагато складнішим методом, але дуже потужним є використання програмного забезпечення, розробленого із можливістю горизонтального масштабування. Наприклад, Apache Cassandra — система управління базами даних, базується на хеш-таблицях, легко масштабується. Cassandra написана на мові програмування Java, код якої компілюється у байт-код і виконується у віртуальній машині, що не дозволить досягти тієї ж швидкодії, як при використанні мов програмування, код на яких компілюється в об'єктний/машинний код.

Інше програмне забезпечення із аналогічним функціоналом — HDFS (Hadoop Distributed File System — розподілена файлова система Hadoop), частина проекту Apache Hadoop (програмне забезпечення для розподілених обчислень). Як і Apache Cassandra, HDFS написаний на Java.

Проаналізувавши методи створення Cloud-систем та вимоги до розроблюваної системи, для розробки програмного забепечення використовуються наступні базові програмні компоненти та технології:

- операційна система Linux (Debian Wheezy)
- мова програмування C
- бібліотека Pthreads
- бібліотека Mhash

Перевагою обраних технологій є відкритість їх вихідного коду та безкоштовність.

Серед операційних систем для розробки програмного забезпечення були розглянуті системи сімейств Microsoft Windows, QNX, Free/Open BSD, Linux.

Можливість використання операційних систем сімейства Microsoft Windows була відхилена через їх високу вартість (особливо серверних версій), а також за надлишковість (графічний інтерфейс користувача не є необхідним для сервера).

Операційна система QNX також була відхилена через її ліцензію: її використання в комерційних цілях є платним. Окрім того, використання операційної системи реального часу не є виправданим для зберігання даних.

Серед UNIX-подібних BSD/Linux систем був обраний Debian Linux, так як він має дуже зручну систему управління пакетами, а також простий встановлювач, що значно спрощує створення та налаштування вузлів у мережі.

2 ЛОГІЧНА СТРУКТУРА ТА ПРИНЦИПИ РОБОТИ МЕРЕЖІ

2.1 Базові принципи

Складовими мережі є вузли — функціональні компоненти, кожен з яких зберігає дані та може обслуговувати клієнтів (приймання, видалення, видача даних).

Вузли можуть бути географічно рознесені на будь-яку відстань.

Між вузлами підтримуються TCP-з'єднання. З'єднані вузли утворюють логічну мережу, топологією якої є “дерево”.

Приклад топології мережі показано на рисунку 1.

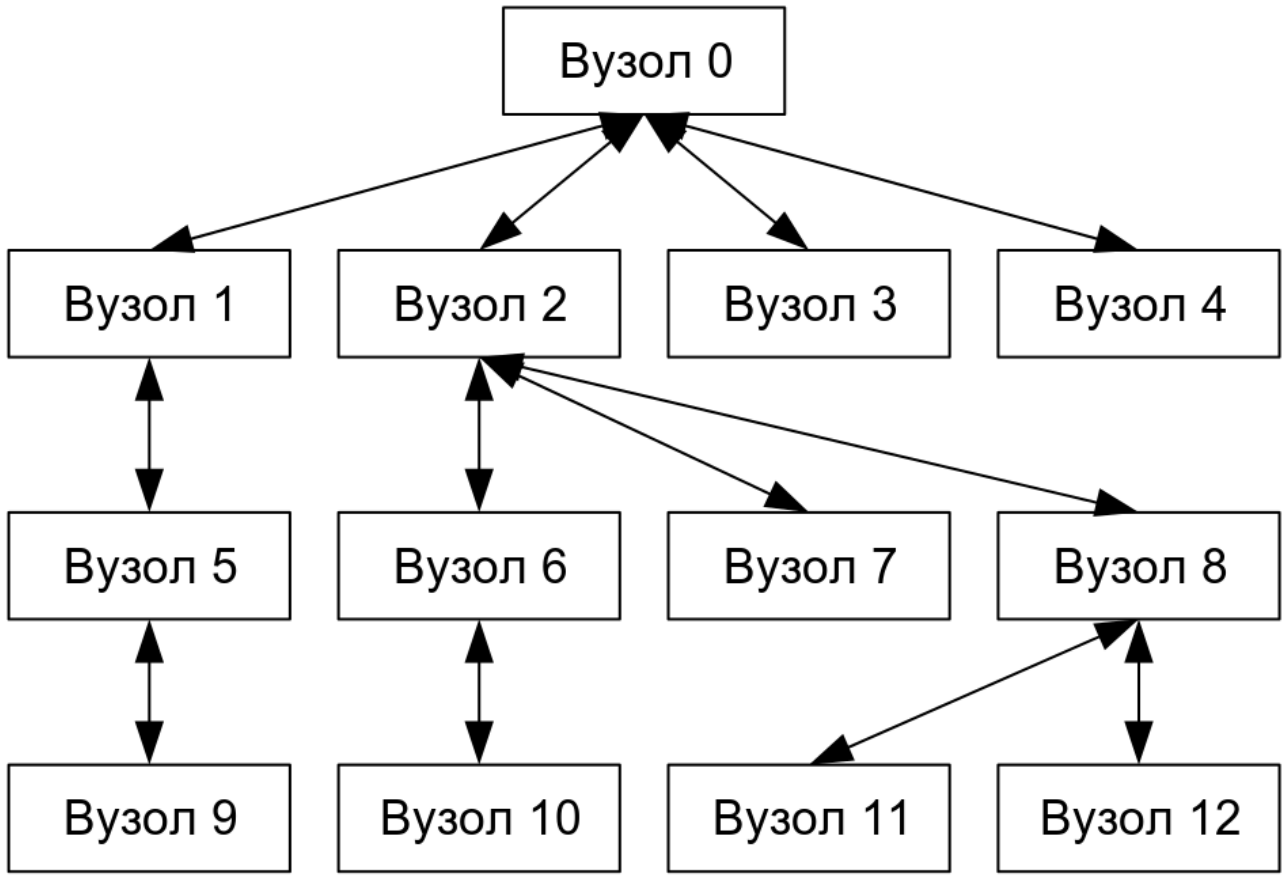

Рисунок 1 — Приклад топології мережі

У мережі не має бути циклів. Для запобігання цього, вузол підключається лише до одного “батьківского” вузла (тобто, вузол не може одразу підключитися до декількох вузлів мережі, він може лише приймати з'єднання дочірніх вузлів).

Кожен вузол може приймати клієнтські з'єднання. Їх кількість не обмежується розроблюваним програмним забезпеченням, лише операційною системою та апаратним забезпеченням (кількість відкритих сокетів, обсяг оперативної пам'яті).

Клієнт (чи інший вузол) намагається постійно підтримувати зв’язок із його батьківським вузлом: заощаджується час на встановленні TCP-з’єднання, окрім того, клієнт, що є бездіяльним, майже не витрачає робочого часу вузла (не генерує подій на обробку).

З’єднання між вузлами в мережі (а також між вузлом і клієнтом) є діалогом: майже на кожен запит, окрім видалення даних, повинна бути реакція батьківського вузла (відповідь на запит даних, приймання даних або розірвання з’єднання).

Робота клієнта із мережею схематично зображена на рисунку 2.

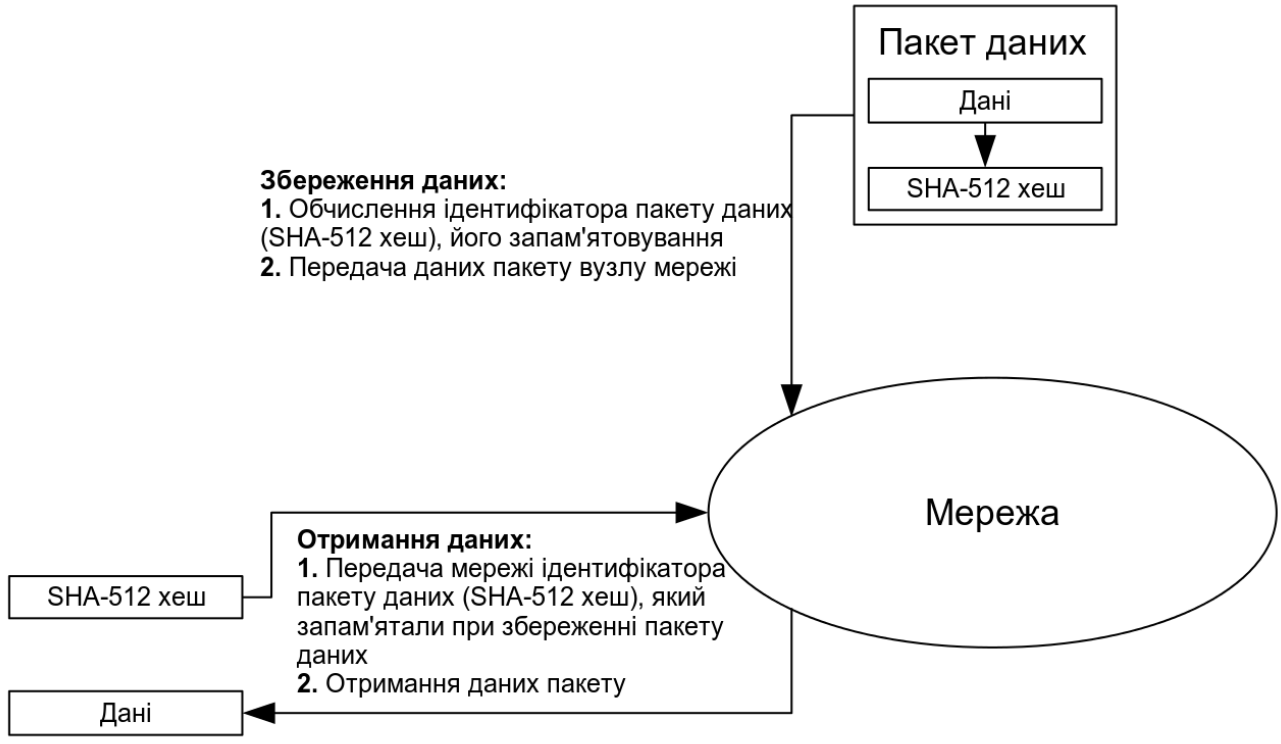

Рисунок 2 — Робота клієнта із мережею

2.2 Авторизація

Кожен вузол в мережі має 3 ключі:

- ключ для читання
- ключ для читання-запису
- ключ вузла

Ключ для читання використовується для отримання даних клієнтом.

Ключ для читання-запису використовується для отримання, додавання та видалення даних.

Ключ вузла використовується для того, щоб ідентифікувати клієнта як вузол, на який можна передати запит, якщо дані для відповіді на нього на поточному вузлі не знайдені.

При підключенні до вузла, клієнт:

- отримує довжину випадкової фрази (1 байт)
- отримає випадкову фразу
- накладає на отриману фразу один із ключів (ключ читання, читання-запису, вузла)
- обчислює SHA-512 хеш отриманої фрази
- надсилає хеш вузлу

Після отримання хешу від клієнта, вузол обчислює:

- хеш від накладання ключа для читання на випадкову фразу, надіслану клієнту; якщо цей хеш співпадає із відповіддю клієнта — клієнт має право читати дані
- інакше — хеш від накладання ключа для читання-запису на випадкову фразу; якщо цей хеш співпадає із відповіддю клієнта — клієнт має право читати і змінювати дані
- інакше — хеш від накладання ключа для вузла на випадкову фразу; якщо цей хеш співпадає із відповіддю клієнта — клієнт є вузлом і йому можна передавати запити, які поточний вузол не може обробити
- якщо ні один із обчислених вузлом хешів не співпав із підповіддю клієнта — з'єднання із клієнтом переривається вузлом (клієнт не має прав працювати із

даними мережі)

Після цього, клієнт виступає в ролі автора випадкової фрази — тобто, клієнт проходить сценарій авторизації у ролі вузла, а вузол, відповідно, в ролі клієнта.

Блок-схема алгоритму авторизації у мережі показана на рисунку 3 (виконання алгоритму, показаного на блок-схемі, виконується два рази, між виконанням — обмін клієнту і вузла ролями).

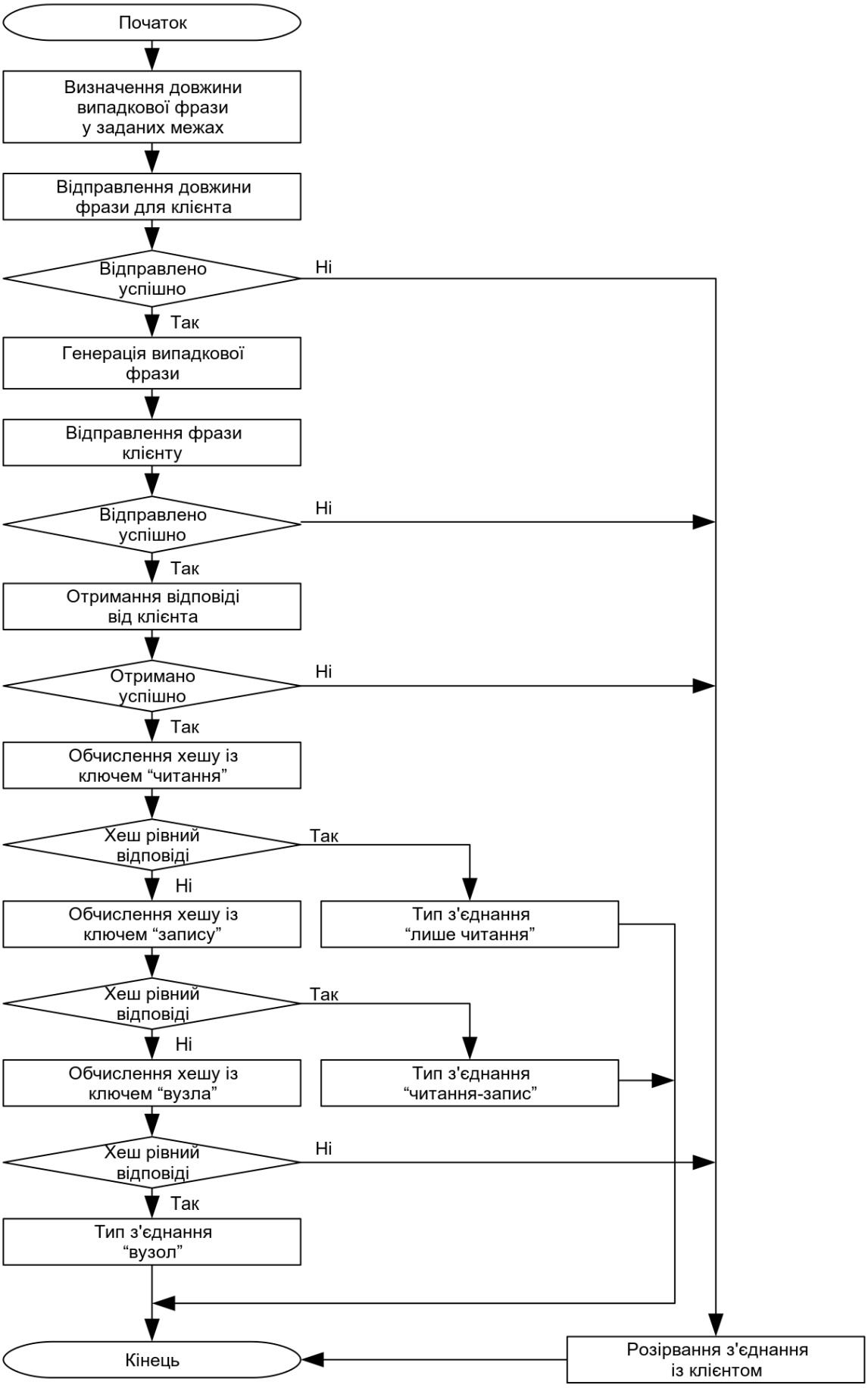

Рисунок 3 — Блок-схема алгоритму авторизації в мережі

2.3 Отримання даних

Кожний пакет даних, збережений у мережі, ідентифікується за SHA-512 хешем його вмісту.

Для отримання даних, клієнт:

- надсилає вузлу команду на видачу даних (1 байт)
- надсилає вузлу хеш-ідентифікатор пакету даних (64 байт)
- отримує дані

Отримання даних відбувається максимум по 255 байтів: вузол надсилає клієнту кількість байтів від 0 до 255 (в 1 байті), які будуть відправлені. Це повторюється, поки частина пакету даних не буде менша за 255 (остання частина пакету даних).

Клієнт, відповідно, отримує 1 байт — розмір частини, потім отримує частину пакету даних, розмір якого відповідає значенню отриманого раніше байту.

Якщо запитуваного пакету даних не знайдено на вузлі мережі, він передає запит на отримання цього пакету усім вузлам, із якими він має з’єднання.

Якщо пакет даних у мережі знайдено не було, клієнту передається 1 байт зі значенням 0 (нульова довжина відповіді).

Блок-схема алгоритму отримання даних із мережі показана на рисунку 4 (алгоритм, який виконується клієнтом), блок-схема алгоритму видачі даних вузлом показана на рисунку 5.

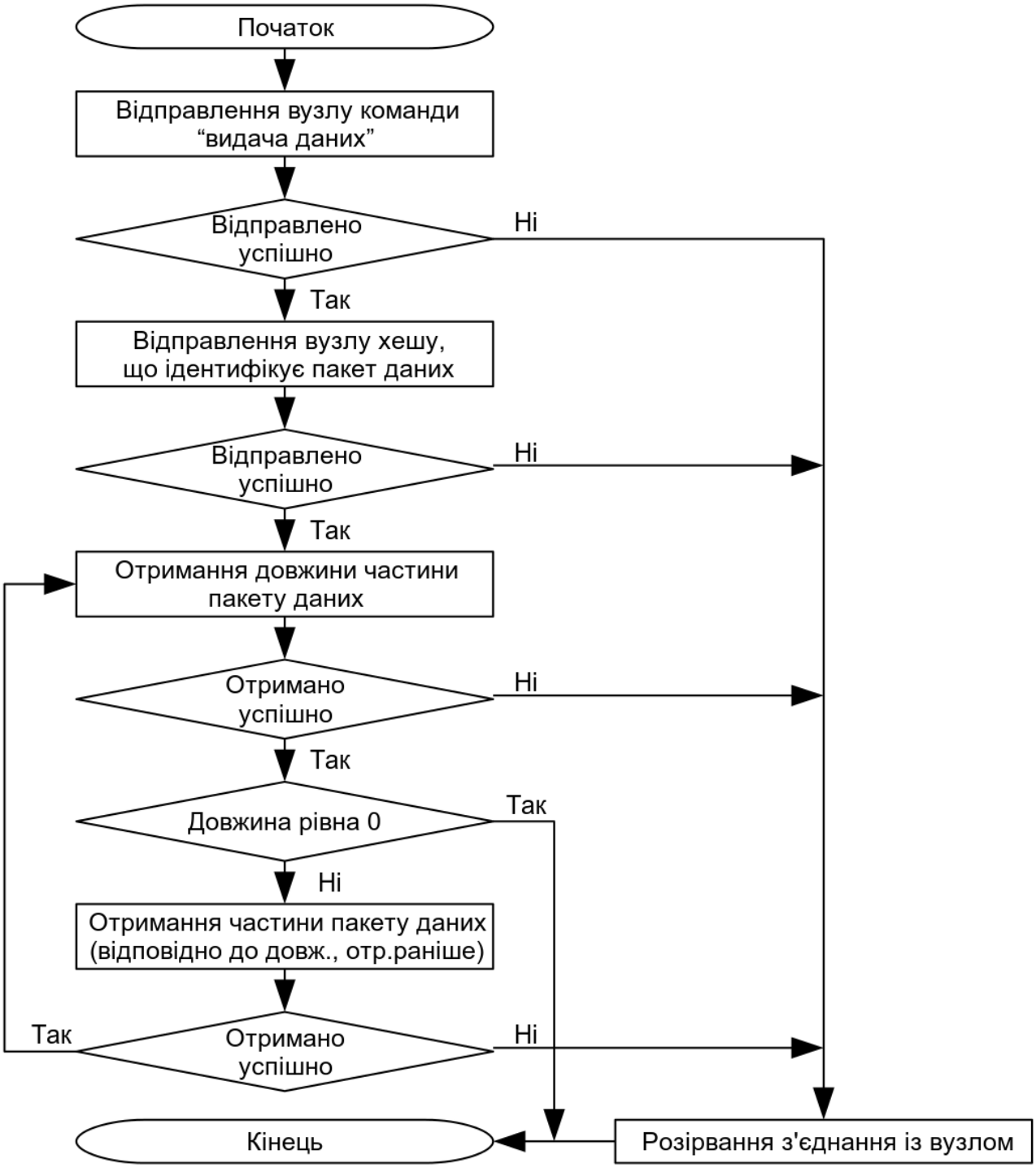

Рисунок 4 — Блок-схема алгоритму отримання даних із мережі

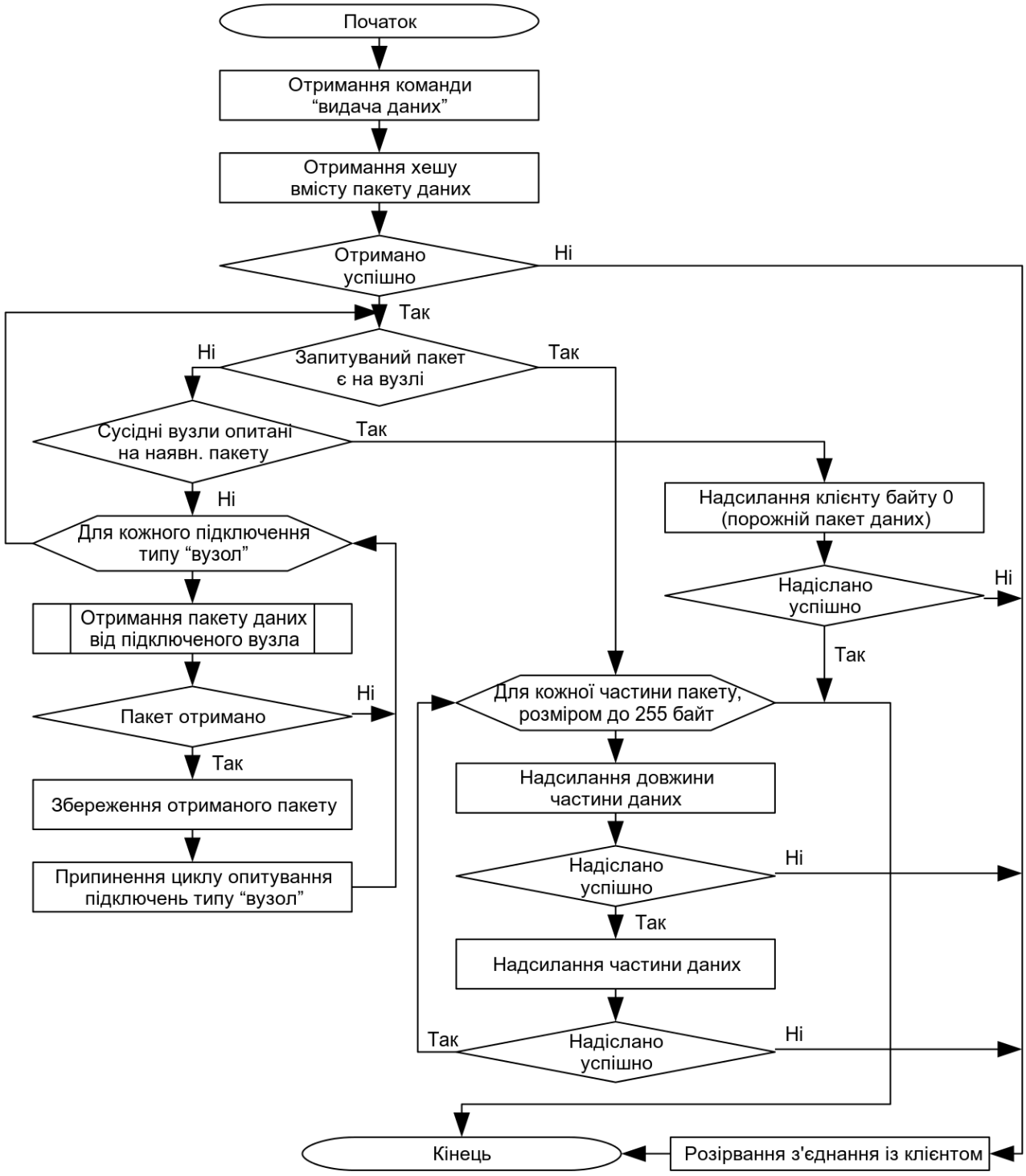

Рисунок 5 — Блок-схема алгоритму видачі даних вузлом

2.4 Збереження даних

Збереження даних ініціюється клієнтом шляхом відправки вузлу 1 байта — команди збереження даних.

Якщо у клієнта немає права на зміну даних у мережі, вузол розриває із ним з'єднання.

Після отримання команди збереження даних, вузол отримує від клієнта SHA-512 хеш пакету даних і починає читати дані частинами розміром до 255 байтів — читання даних вузлом відбувається аналогічно п. 3.3 із відмінністю, що джерелом даних є клієнт, а приймачем — вузол.

Після отримання усього пакету даних для збереження, вузол обчислює його SHA-512 хеш.

Якщо обчислений хеш не співпадає із отриманим від клієнта на початку процедури збереження даних, сервер відправляє клієнту 1 байт зі значенням 0 (дані не збережено).

Якщо обчислений хеш співпадає із надісланим клієнтом, вузол і зберігає дані і відправляє клієнту 1 байт зі значенням 1 (дані успішно збережено).

Блок-схема алгоритму збереження даних у мережі показана на рисунку 6.

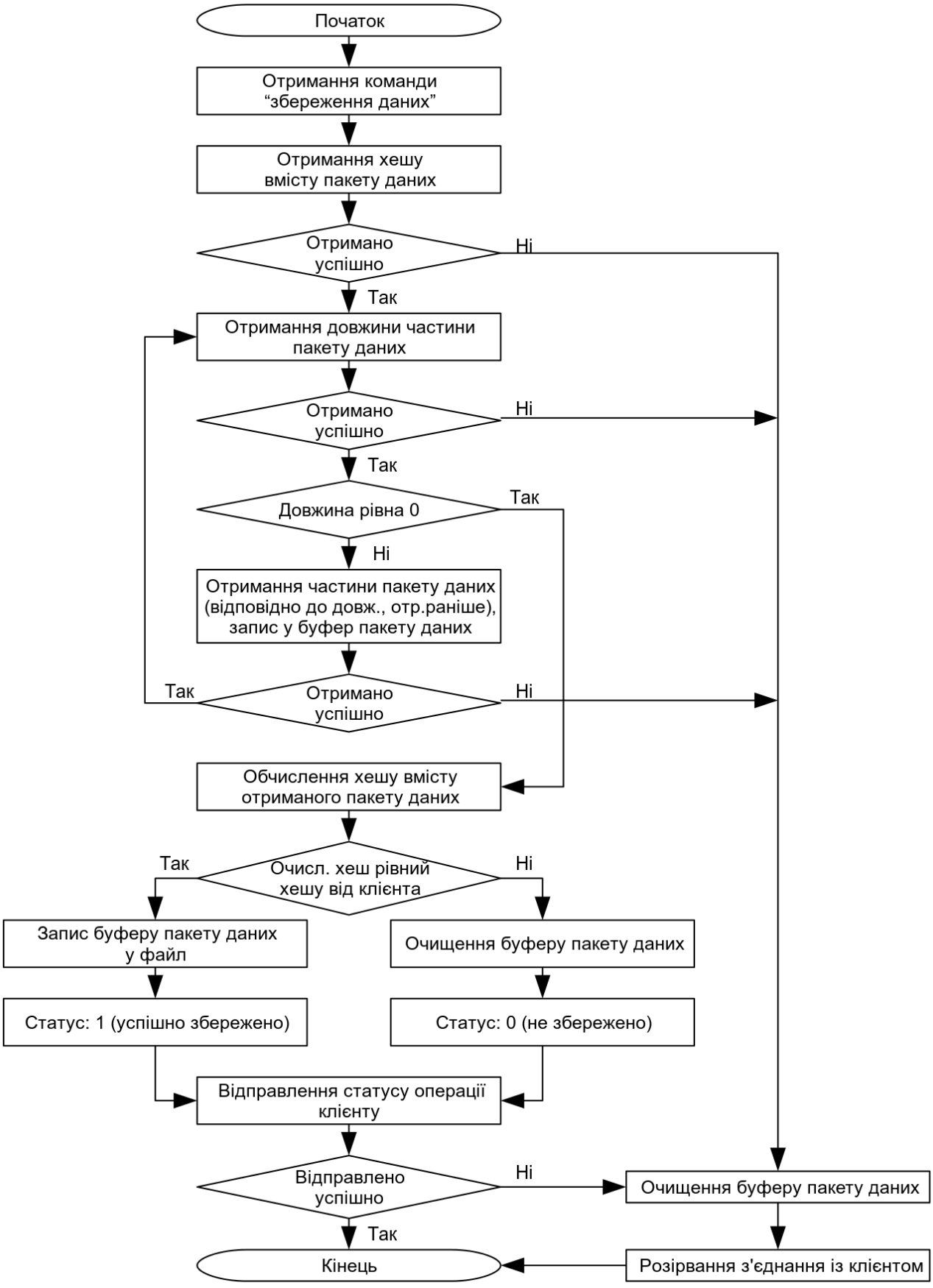

Рисунок 6 — Блок-схема алгоритму збереження даних у мережі

2.5 Видалення даних

Ініціюється клієнтом шляхом відправлення вузлу відповідної команди і хешу пакету даних для видалення.

Якщо у клієнта немає прав на зміну даних в мережі, вузол розриває із ним з'єднання.

Якщо у клієнта є права на зміну даних в мережі — вузол шукає дані із вказаним клієнтом хешем. Якщо пакет даних знайдено — він видаляться.

Клієнту результат виконання команди не повідомляється.

2.6 Відмовостійкість системи

Імовірними причинами відмови в обслуговуванні запиту клієнта є:

- нескінченна кількість вузлів у мережі — на практиці недосяжна
- брак вільної оперативної пам'яті на вузлі системи, до якого підключений клієнт (для фіксування запиту повинна бути створена подія, що зберігається в оперативній пам'яті) — на практиці не є великою проблемою, так як ядро операційної системи (Linux) може використовувати технологію свопінгу (англ. swapping — збереження даних із оперативної пам'яті на диск і відновлення їх в оперативній пам'яті з диску в разі потреби), що приводить до зниження швидкості обробки запитів, але не до повної відмови
- брак дискового простору, відведеного для свопінгу — малоімовірний варіант, але може трапитися при перенавантаженні клієнтами вузла мережі
- брак дискового простору для збереження даних — у поточній архітектурі системи не передбачено передачі запитів на збереження запитів іншим вузлам мережі (підмережам), але при подальшому розвитку архітектури системи і розробці подібного алгоритму можна зменшити імовірність появи такої ситуації
- відмова програмного забезпечення вузла внаслідок помилки, допущеної при його написанні — існує велика імовірність прояву помилок в програмі, яка не була протестована великою кількістю користувачів, хоча після написання (при впровадженні в роботу) програмного забезпечення кількість помилок

вважається нульовою

- відмова операційної системи або програмного забезпечення, на використанні якого базується програмне забезпечення вузла — наймалоімовірніша ситуація, так як використовувана операційна система та базове програмне забезпечення тестує велика кількість людей з усього світу і помилки досить швидко виправляються

Повна відмова в обслуговуванні клієнта відбувається лише у разі відмови вузла, до якого безпосередньо підключений клієнт. В інших випадках, коли запит клієнта передається іншим вузлам мережі, відмова одного із вузлів на шляху даних розцінюється як порожній результат (запитуваних даних не знайдено).

2.7 Узагальнений опис мережі

Кожен вузол спроектованої комп'ютерної системи можна розцінювати як інтерфейс доступу до її функцій.

Тож з'єднання виду “вузол-вузол” можна інтерпретувати як з'єднання типу “мережа-мережа”. Тобто, можна говорити про те, що основна мережа складається не із вузлів, а із підмереж. Кожна підмережа, в свою чергу, також є з'єднанням підмереж, але нижчого рівня.

При пошуку даних, такий підхід дозволяє абстрагуватися від опиту кожного вузла основної мережі, опитуючи натомість кожну її підмережу через вузол, який є інтерфейсом для цієї підмережі.

Система спроектована таким чином, щоб гарантувати видачу запитаних даних, якщо хоча б один її вузол має такі дані. Це значно підвищує імовірність отримання даних із мережі, якщо вони колись були в ній збережені, але сильно впливає на швидкість її роботи, так як для отримання даних можуть бути опитані всі вузли мережі. А так як кількість вузлів у мережі теоретично необмежена, то і час очікування обслуговування запиту клієнта може бути нескінченним. Це є одним із випадків, коли запит клієнта не буде оброблено. Але, на практиці нескінченна кількість вузлів у мережі не є досяжною, тож можна говорити лише про великий, але не нескінченний час очікування обробки запиту.

Для зменшення кількості подібних “довгих” запитів, дані при проходженні

шляху від вузла, на якому вони були знайдені, до клієнта копіюються на кожному проміжному вузлі, який має місце для їх збереження. Тобто, при повторному запиті даних, швидкість їх видачі буде значно вищою за швидкість обробки їх першого запиту, так як дані вже будуть збережені на найближчому до клієнта вузлі системи (кількість передач запиту від вузла до вузла прямує до нуля).

Загалом, розроблювану ком'ютерну систему можна описати як систему масового обслуговування із очікуванням і теоретично необмеженим буфером для збереження черги запитів, що працює за принципом FCFS (англ. first come, first served — першим надійшов, першим обслужено) та є відносно відмовостійкою (відмова в обслуговуванні запиту залежить лише від вузла, якому клієнт адресує запит). Відмова вузла на шляху проходження даних є повністю допустимою (вузли і підмережі можуть неконтрольовано підключатися і відключатися від основної мережі), тож така відмова розцінюється як відсутність результату, про що клієнт повідомляється.

3 ПРИНЦИПИ РОБОТИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ВУЗЛА

При запуску програмного забезпечення вузла мережі, проходить його ініціалізація:

- якщо програмі в якості аргумента була передана IP-адреса – вона намагається підключитися до вказаної адреси на стандартному порті мережі. Якщо з'єднання було прийнято, поточний вузол (на якому відбувається ініціалізація програмного забезпечення) проходить процедуру авторизації (описану в п. 3.2). При успішній процедурі авторизації, створене з'єднання зберігається новим вузлом у списку усіх з'єднань із типом «з'єднання із вузлом»
- створюється TCP-сокет, здатний приймати вхідні з'єднання (англ. listening socket)
- створюються «виконавці» (англ. workers) для подій вузла
- на сокеті починається очікування вхідних з'єднань

Алгоритм роботи програмного забезпечення вузла базується на опрацюванні декількома «виконавцями» подій, що генеруються при взаємодії клієнтів або інших вузлів із поточним вузлом.

Будь-що є подією: отримання нового з'єднання вузлом, отримання вузлом команди на виконання (отримання запиту клієнта на опрацювання), переадресація запиту одному із сусідніх вузлів тощо.

«Виконавці» - це треди (англ. threads, потоки), які знаходяться або в стані опрацювання події, або в стані очікування подій.

Для очікування подій використовуються такі засоби бібліотеки pthreads, як мьютекси (англ. mutexes) та змінні умов (англ. condition variables).

За допомогою функції pthread_cond_wait, потоки переводяться у «сплячий стан», в якому вони майже не потребують ресурсів процесора.

При появі нової події у черзі подій (англ. event poll), за допомогою функції pthread_cond_signal змінна умови черги подій стає активною, що призводить до активізації одного із вільних «виконавців».

Особливістю pthread_cond_signal є те, що при зайнятості усіх потоків, цей сигнал буде втрачено (на нього не буде ніякої реакції). Для розробленої програми це не є критичною ситуацією, так як по завершенню виконання задачі, потік спочатку перевіряє вершину черги подій. Якщо вершина порожня – потік зупиняється в очікуванні сигналу про появу нової події. Інакше, якщо вершина не порожня, потік забирає верхню подію на виконання. Тож простою ресурсів через пропуск сигнала про появу нової події неможливий.

Черга подій реалізована на основі зв'язного замкненого списку (схематично зображена на рисунку 7).

Кожна подія є структурою, що має такі поля:

- дескриптор сокета події (із яким необхідно виконати якусь дію, базуючись на типі події)
- тип події (визначає дію, яку необхідно виконати при обробці події)
- ліва подія (попередня подія)
- права подія (наступна подія)

Можлива ситуація, коли у списку події є лише одна подія (цей випадок схематично показано на рисунку 8). В такому разі, її поля «попередня» та «наступна» подія вказують на неї саму. При обробці такої події (із рівними вказівниками на сусідні події), необхідно обнулити вершину списку (забирається останній елемент списку і він стає порожнім).

Нові події поміщаються у список справа від вершини (схематично показано на рисунку 9); при отриманні події для опрацювання зі списку, забирається подія із його вершини, а на її місце ставиться вказівник події зліва від вершини (схематично показано на рисунку 10).

Таким чином, створюється черга подій, що працює за принципом FIFO/FCFS (англ. First In — First Out, First Come — First Served, перший надійшов — перший обслужений).

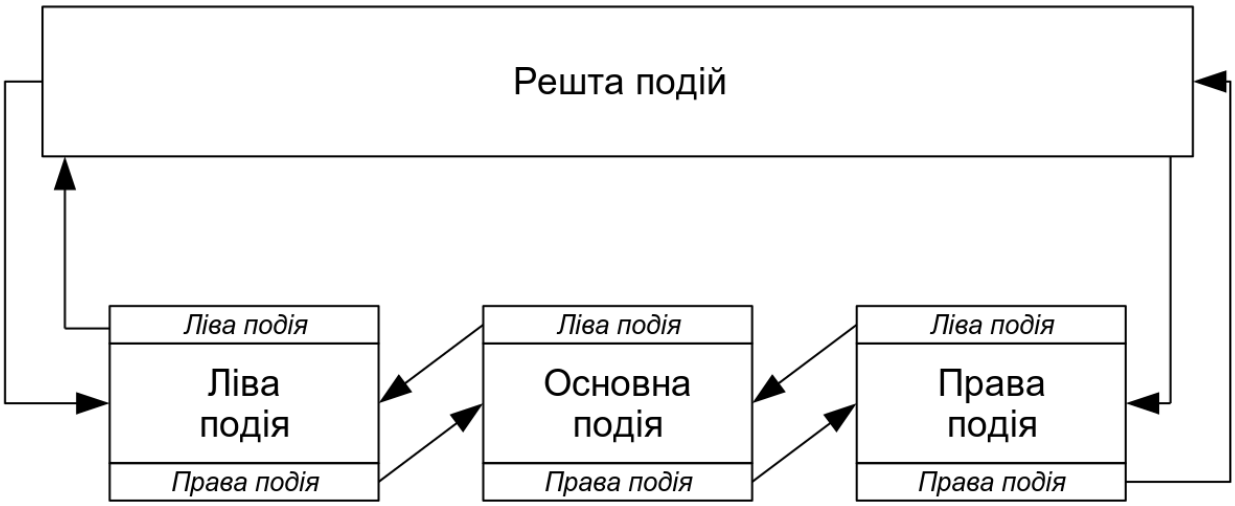

Рисунок 7 — Схематичне зображення черги подій

Рисунок 8 — Схематичне зображення черги подій, що має лише один елемент

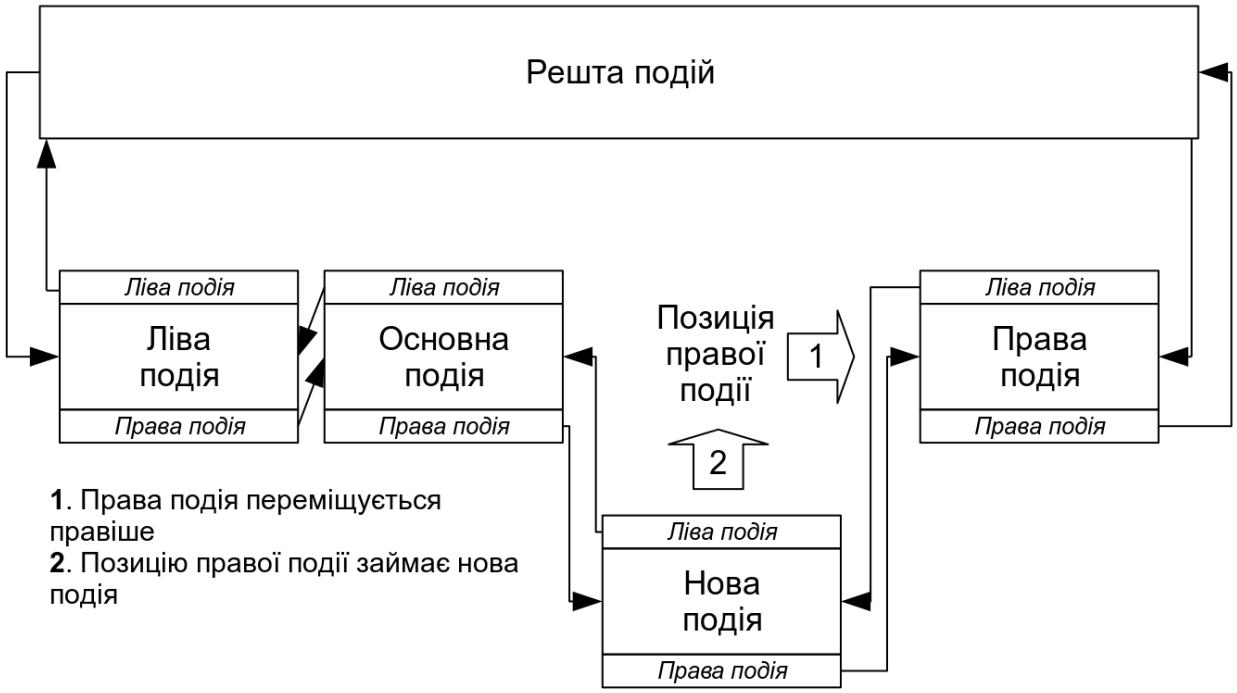

Рисунок 9 — Додавання події у чергу

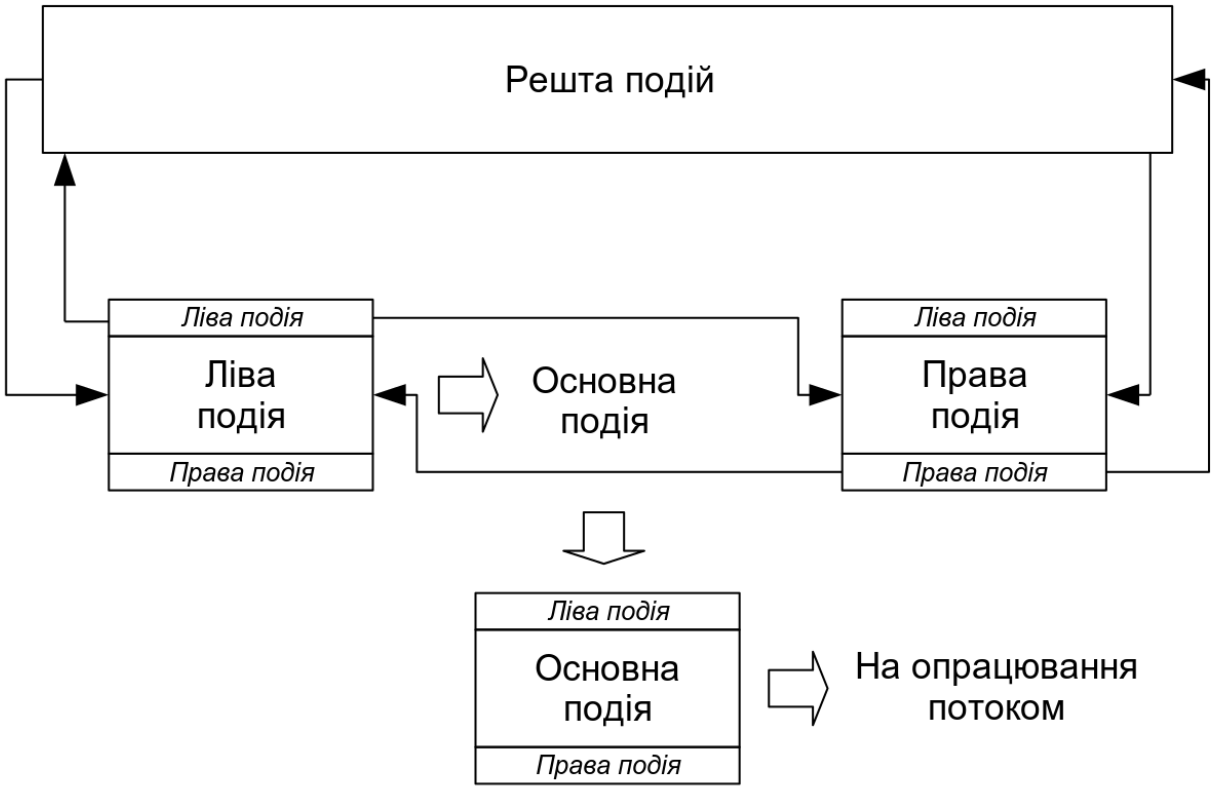

Рисунок 10 — Забирання події із черги для обробки

Аналогічним чином зберігаються з’єднання, які підтримує вузол. Ця черга (черга з’єднань) реалізована перш за все для зберігання IP-адрес сусідніх вузлів для передачі їх новим підключеним вузлам: це дає шанс клієнту (або вузлу, для якого поточний вузол є батьківським) підключитися до одного із сусідніх вузлів при відмові серверного (батьківського) вузла.

Окрім цього, така черга з'єднань дозволить при подальшому вдосконаленні програмного забезпечення вузла системи реалізувати адміністративне керування з'єднаннями вузла під час його роботи. Наприклад, можна реалізувати адміністративну команду, яка буде закривати певний сокет або розривати з'єднання із певним клієнтом по його IP-адресі.

Підсумовуючи опис роботи програмного забезпечення вузла мережі, можна виділити такі основні принципи:

- кожен вузол має декілька виконавчих процесів-конвеєрів, кожен із яких може опрацьовувати одну подію в один момент часу або чекати на появу події для опрацювання
- виконавчі процеси забезпечують осблуговування декількох запитів (виконання декількох подій) одночасно, максимальна кількість подій (запитів), що обслуговуються одночасно, рівна кількості робочих процесів-конвеєрів
- кожен вузол має чергу подій, в якій реєструються запити, що не можуть бути обслужені моментально (коли усі виконавці зайняті опрацюванням інших подій)
- балансування навантаження на вузол зводиться до розподілу подій із черги між процесами-конвеєрами
- якщо у вузла немає даних, які були запитані клієнтом, він переадресовує запит клієнта іншим вузлам мережі, зберігаючи у себе дані для подальшого використання, якщо вони були знайдені у мережі (це дозволяє значно скоротити час обслуговування повторного запиту одного і того ж пакету даних у порівнянні із першим)
- система не розрахована на видалення даних в усій мережі — при запиті

видалення даних, пакет даних видаляється лише на вузлі, до якого цей запит надійшов; при подальшому розвитку програмного забезпечення мережі можна реалізувати переадресацію подібного запиту усій мережі

- програмне забезпечення вузла має функціонал, який може бути використаний при подальшому його розвитку (наприклад, список з'єднань вузла для реалізації адміністративних команд)
- вузол не відстежує маршрут, яким проходять дані при їх пошуку для клієнта, це зроблено для того, щоб зменшити розмір пакету даних при його пересилці між вузлами (дані про маршрут пакету даних не зберігаються і не передаються між вузлами)
- відмова одного із вузлів на шляху даних у мережі інтерпретується не як відмова системи, а як відсутність результату — система спроектована таким чином, щоб бути динамічною, тобто, підключення і відключення вузлів і підмереж від основної системи є непрогнозованим і сприймається, як належне

Принцип роботи програмного забезпечення вузла схематично представлено на рисунку 11. Блок-схему роботи конвеєру (робочого потоку вузла) показано на рисунку 12.

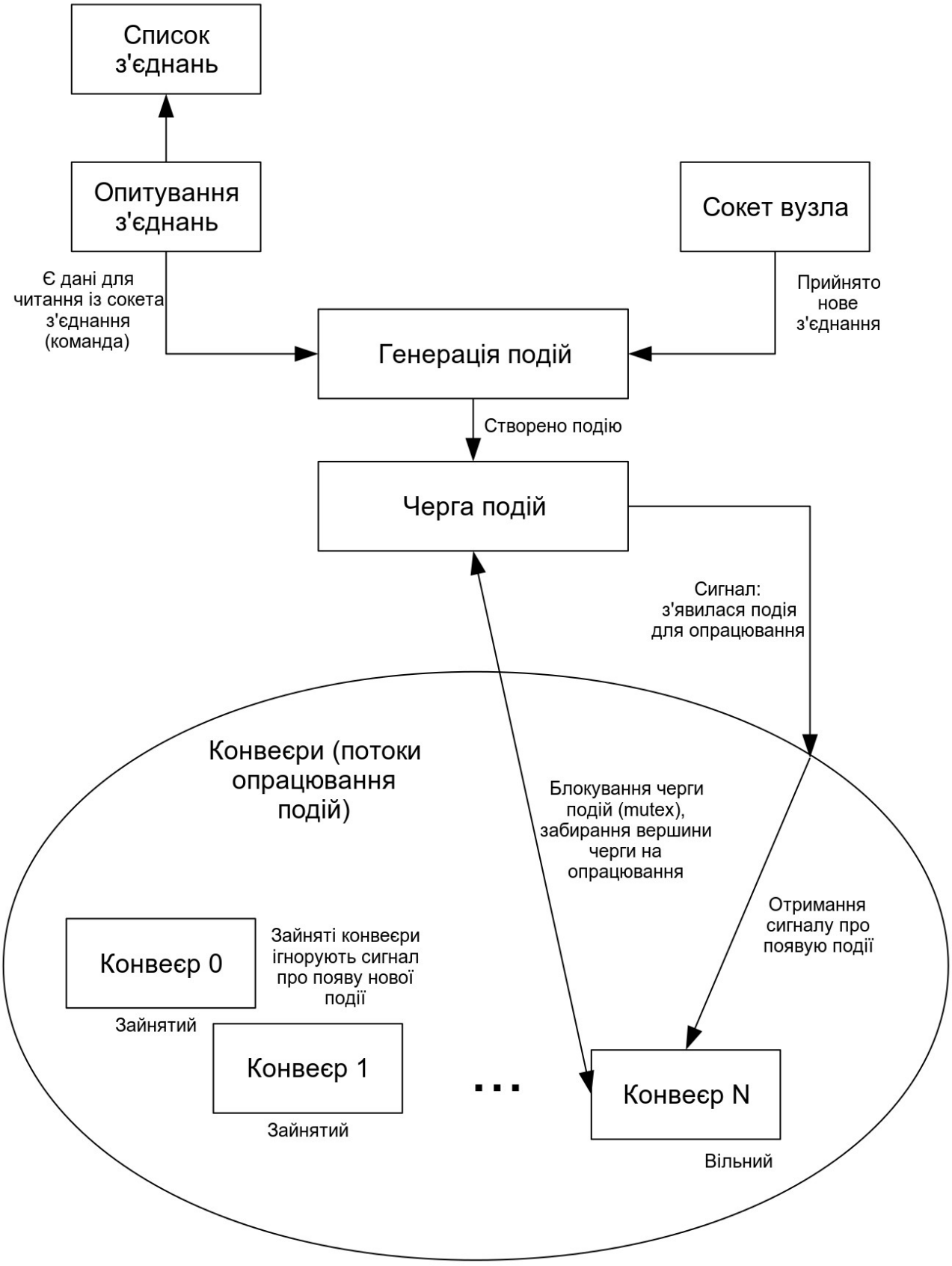

Рисунок 11 — Принцип роботи програмного забезпечення вузла

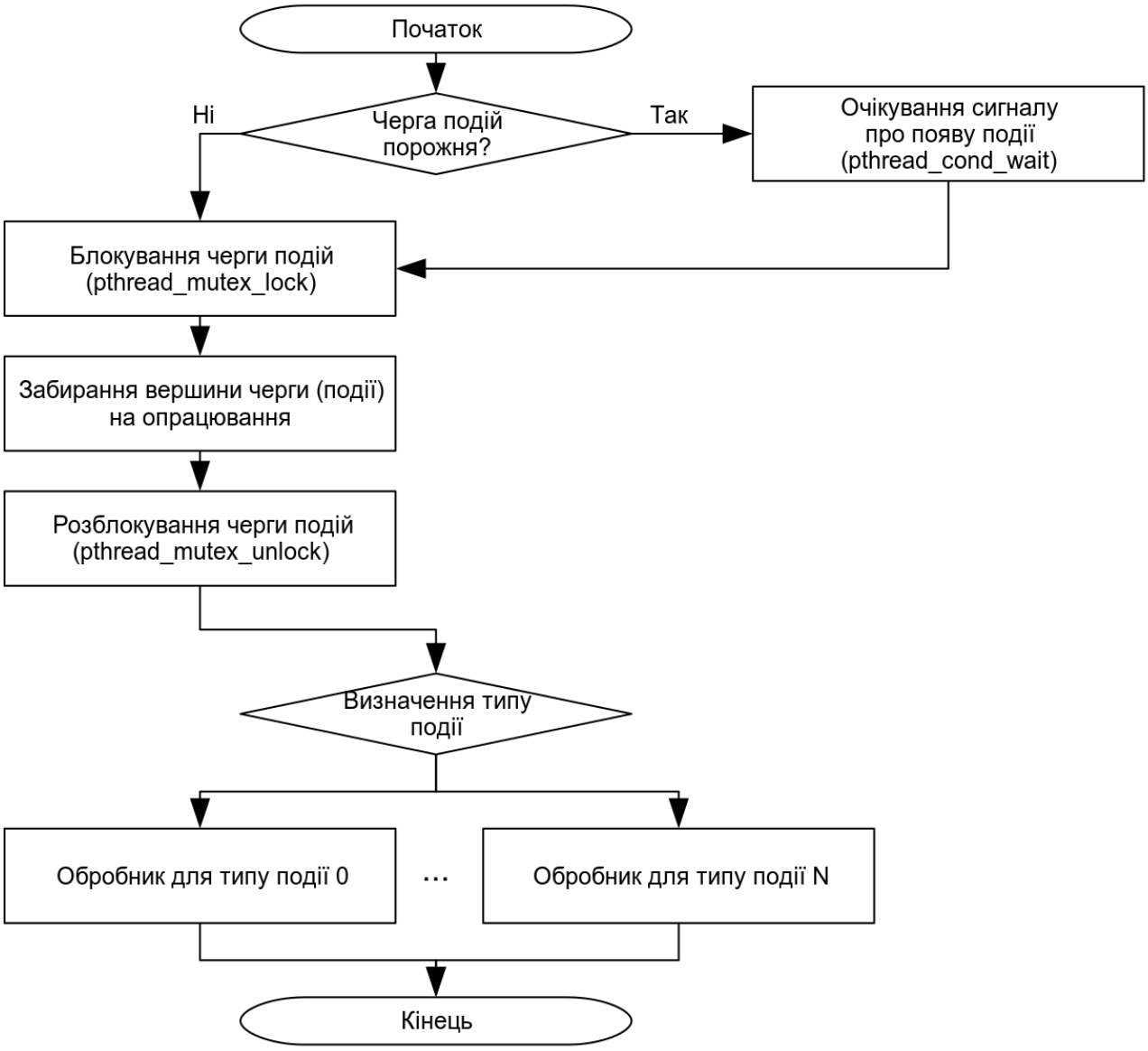

Рисунок 12 — Блок-схема алгоритму роботи потоку вузла

4 ФРАГМЕНТИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ВУЗЛА МЕРЕЖІ

4.1 Підключення необхідних бібліотек

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <unistd.h>
#include <pthread.h>
#include <mhash.h>
#include <string.h>
#include <sys/stat.h>
```

4.2 Оголошення необхідних констант

```
#define NODE_DOMAIN  INADDR_ANY
#define NODE_PORT    4448
#define NODE_BACKLOG 300
#define NODE_WORKERS 300

#define NODE_READ_KEY  "01234567890123456789012345678901"
#define NODE_WRITE_KEY "01234567890123456789012345678902"
#define NODE_NODE_KEY  "01234567890123456789012345678903"

#define NODE_CLIENT_SOCKET_TIMEOUT 30

#define NODE_MIN_KEY_LENGTH 32
#define NODE_MAX_KEY_LENGTH 256

#define NODE_CHALLENGE_MIN_LENGTH 32
#define NODE_CHALLENGE_MAX_LENGTH 255

#define NODE_HASH_LENGTH 64

#define NODE_CONNECTION_READONLY  1
#define NODE_CONNECTION_READWRITE 2
#define NODE_CONNECTION_NODE      3

#define NODE_DATA_PATH "./"
```

4.3 Ініціалізація програми вузла

```
typedef struct node_t {
        event_t         *event;
        pthread_mutex_t *event_mutex;
        pthread_cond_t  *event_condition;
        connection_t    *connection;
        pthread_mutex_t *connection_mutex;

        char *read_key;
        int   read_key_length;

        char *write_key;
        int   write_key_length;

        char *node_key;
        int   node_key_length;

} node_t;
```

```
node_t *create_node( char *read_key, char *write_key, char *node_key ) {

	if( 0 == read_key ) {
		fprintf( stderr, "No read-key provided\n" );
		return 0;
	};

	if( 0 == write_key ) {
		fprintf( stderr, "No write-key provided\n" );
		return 0;
	};

	if( 0 == node_key ) {
		fprintf( stderr, "No node-key provided\n" );
		return 0;
	};

	int read_key_length  = strlen( read_key );
	int write_key_length = strlen( write_key );
	int node_key_length  = strlen( node_key );

	if( NODE_MIN_KEY_LENGTH > read_key_length ) {
		fprintf( stderr, "Read key should be at least %i characters long", NODE_MIN_KEY_LENGTH
);
		return 0;
	};

	if( NODE_MAX_KEY_LENGTH < read_key_length ) {
		fprintf( stderr, "Read key should not be longer than %i characters",
NODE_MAX_KEY_LENGTH );
		return 0;
	};

	if( NODE_MIN_KEY_LENGTH > write_key_length ) {
		fprintf( stderr, "Write key should be at least %i characters long",
NODE_MIN_KEY_LENGTH );
		return 0;
	};

	if( NODE_MAX_KEY_LENGTH < write_key_length ) {
		fprintf( stderr, "Write key should not be longer than %i characters",
NODE_MAX_KEY_LENGTH );
		return 0;
	};

	if( NODE_MIN_KEY_LENGTH > node_key_length ) {
		fprintf( stderr, "Node key should be at least %i characters long", NODE_MIN_KEY_LENGTH
);
		return 0;
	};

	if( NODE_MAX_KEY_LENGTH < node_key_length ) {
		fprintf( stderr, "Node key should not be longer than %i characters",
NODE_MAX_KEY_LENGTH );
		return 0;
	};

	node_t *node = malloc( sizeof( node_t ) );

	if( 0 == node ) {
		perror( "Failed to allocate memory for node" );
		return 0;
	};

	node->event_mutex = malloc( sizeof( pthread_mutex_t ) );

	if( 0 == node->event_mutex ) {
		perror( "Failed to allocate memory for node event mutex" );
		free( node );
		return 0;
	};

	node->event_condition = malloc( sizeof( pthread_cond_t ) );
```

```
        if( 0 == node->event_condition ) {
                perror( "Failed to allocate memory for node event condition" );
                pthread_mutex_destroy( node->event_mutex );
                free( node->event_mutex );
                free( node );
                return 0;
        };

        node->connection_mutex = malloc( sizeof( pthread_mutex_t ) );

        if( 0 == node->connection_mutex ) {
                perror( "Failed to allocate memory for node connection mutex" );
                pthread_mutex_destroy( node->event_mutex );
                pthread_cond_destroy( node->event_condition );
                free( node->event_mutex );
                free( node->event_condition );
                free( node );
                return 0;
        };

        node->event      = 0;
        node->connection = 0;

        node->read_key   = read_key;
        node->write_key  = write_key;
        node->node_key   = node_key;

        node->read_key_length   = read_key_length;
        node->write_key_length  = write_key_length;
        node->node_key_length   = node_key_length;

        return node;

};

int init_node_socket( int domain, int port, int backlog ) {

        int node_socket = socket( AF_INET, SOCK_STREAM, 0 );

        if( -1 == node_socket ) {
                perror( "Failed to create node socket" );
                return -1;
        };

        struct sockaddr_in node_address;

        node_address.sin_family      = AF_INET;
        node_address.sin_port        = port;
        node_address.sin_addr.s_addr = domain;

        if( -1 == bind( node_socket, ( struct sockaddr * ) &node_address, sizeof( node_address ) ) ) {
                perror( "Failed to bind node socket" );
                close( node_socket );
                return -1;
        };

        if( -1 == listen( node_socket, backlog ) ) {
                perror( "Failed to listen on node socket" );
                close( node_socket );
                return -1;
        };

        return node_socket;

};
```

```
int init_master_socket( int domain, int port ) {

	int master_socket = socket( AF_INET, SOCK_STREAM, 0 );

	if( -1 == master_socket ) {
		perror( "Failed to create master socket" );
		return -1;
	};

	struct sockaddr_in master_address;

	master_address.sin_family      = AF_INET;
	master_address.sin_port        = port;
	master_address.sin_addr.s_addr = domain;

	if( -1 == connect( master_socket, ( struct sockaddr * ) &master_address,
sizeof( master_address ) ) ) {
		perror( "Failed to connect to master node" );
		close( master_socket );
		return -1;
	};

	return master_socket;

};

pthread_t *init_workers( int count, void *( worker_routine )( void * ), node_t *node ) {

	if( 1 > count ) {
		fprintf( stderr, "create_workers: count == %i, count < 1\n", count );
		return 0;
	};

	if( 0 == node ) {
		fprintf( stderr, "create_workers: node == 0");
		return 0;
	};

	if( 0 != pthread_setcancelstate( PTHREAD_CANCEL_ENABLE, 0 ) ) {
		perror( "Failed to make threads (workers) cancelable" );
		return 0;
	};

	if( 0 != pthread_setcanceltype( PTHREAD_CANCEL_ASYNCHRONOUS, 0 ) ) {
		perror( "Failed to make threads (workers) cancelable asynchronously" );
		return 0;
	};

	pthread_t *workers = malloc( sizeof( pthread_t ) * count );

	int i = 0;

	for( ; i < count; i++ ) {

		if( 0 != pthread_create( &workers[i], 0, worker_routine, node ) ) {

			perror( "Failed to create thread (worker)" );

			i--;

			for( ; i >= 0; i-- ) {
				pthread_cancel( workers[i] );
			};

			free( workers );

			return 0;

		};

	};

	return workers;

};
```

```
int main( int argc, char **argv, char **env ) {

	node_t *node = create_node( NODE_READ_KEY, NODE_WRITE_KEY, NODE_NODE_KEY );

	if( 0 == node ) {
		return 1;
	};

	pthread_t *workers = init_workers( NODE_WORKERS, worker_routine, node );

	if( 0 == workers ) {
		return 1;
	};

	int node_socket = init_node_socket( NODE_DOMAIN, htons( NODE_PORT ), NODE_BACKLOG );

	if( -1 == node_socket ) {
		return 1;
	};

	for( ; ; ) {

		int client_socket = accept( node_socket, 0, 0 );

		if( -1 == client_socket ) {
			perror( "Failed to accept client connection" );
			continue;
		};

		connection_t *connection = create_connection( client_socket, 0 );

		if( 0 == connection ) {
			continue;
		};

		event_t *event = create_event( client_socket, 0 );

		if( 0 == event ) {
			destroy_connection( node, client_socket );
			continue;
		};

		if( -1 == add_connection( node, connection ) ) {
			close( node_socket );
			return 1;
		};

		if( -1 == enqueue_event( node, event ) ) {
			close( node_socket );
			return 1;
		};

	};

	return 0;

};
```

4.4 Процедура конвеєру для опрацювання подій

```
void *worker_routine( void *node_pointer ) {

	if( 0 == node_pointer ) {
		fprintf( stderr, "worker_routine: node_pointer == 0\n" );
		return 0;
	};

	node_t *node = ( node_t * ) node_pointer;

	for( ; ; ) {

		event_t *event = dequeue_event( node );

		int connection_type = get_connection_type( node, event->socket );

		if( -2 == connection_type ) {
			return node_pointer;
		};

		if( -1 == connection_type ) {
			free( event );
			continue;
		};

		if( -1 == event->socket ) {
			free( event );
			continue;
		};

		switch( event->type ) {
			case 0: client_authentification( node, event ); break; // auth client-node
			case 1: break; // auth node-node
			case 2: client_get_chunk( node, event ); break; // give chunk to client
			case 3: client_put_chunk( node, event ); break; // put chunk
			case 4: client_drop_chunk( node, event ); break; // drop chunk
			case 5: client_read_command( node, event ); break; // read command from client
			default: destroy_connection( node, event->socket );
		};

		free( event );

	};

	return node_pointer;

};
```

4.5 Реалізація черги подій

4.5.1 Реалізація структури події

```
typedef struct event_t {
	int socket;
	int type;
	struct event_t *left;
	struct event_t *right;
} event_t;
```

4.5.2 Функція створення події

```
event_t *create_event( int socket, int type ) {

	if( -1 == socket ) {
		fprintf( stderr, "create_connection: socket == -1\n" );
	};

	event_t *event = malloc( sizeof( event_t ) );

	if( 0 == event ) {
		perror( "Failed to allocate memory while creating event" );
		return 0;
	};

	event->socket = socket;
	event->type   = type;

	event->left   = 0;
	event->right  = 0;

	return event;

};
```

4.5.3 Процедура знищення черги подій

```
int destroy_events( node_t *node, int socket ) {

	if( 0 == node ) {
		fprintf( stderr, "destroy_events: node == 0" );
		return 0;
	};

	if( -1 == socket ) {
		fprintf( stderr, "destroy_events: socket == -1" );
		return 0;
	};

	if( 0 == node->event_mutex ) {
		fprintf( stderr, "destroy_events: node->event_mutex == 0" );
		return 0;
	};

	if( 0 != pthread_mutex_lock( node->event_mutex ) ) {
		perror( "Failed to lock event mutex while destroying events" );
		return -1;
	};

	int count = 0;

	if( node->event ) {

		event_t *current = node->event->right;

		do {

			if( current->socket == socket ) {

				if( current->left == current->right ) {

					free( node->event );
					node->event = 0;
					count++;
					break;

				} else {

					current->left->right = current->right;
					current->right->left = current->left;

					free( current );

				};

				count++;

			};

		} while( current != node->event );

	};

	if( 0 != pthread_mutex_unlock( node->event_mutex ) ) {
		perror( "Failed to unlock event mutex while destroying event" );
		return -1;
	};

	return count;

};
```

4.5.4 Процедура додавання події у чергу

```
int enqueue_event( node_t *node, event_t *event ) {

        if( 0 == node ) {
                fprintf( stderr, "enqueue_event: node == 0" );
                return 0;
        };

        if( 0 == event ) {
                fprintf( stderr, "enqueue_event: event == 0" );
                return 0;
        };

        if( 0 == node->event_mutex ) {
                fprintf( stderr, "enqueue_event: node->event_mutex == 0" );
                return 0;
        };

        if( 0 == node->event_condition ) {
                fprintf( stderr, "enqueue_event: node->event_condition == 0" );
                return 0;
        };

        if( 0 != pthread_mutex_lock( node->event_mutex ) ) {
                perror( "Failed to lock event mutex while enqueueing event" );
                return -1;
        };

        if( 0 == node->event ) {
                event->left  = event;
                event->right = event;
                node->event  = event;
        } else {
                event->left              = node->event;
                event->right             = node->event->right;
                node->event->right->left = event;
                node->event->right       = event;
        };

        if( 0 != pthread_cond_signal( node->event_condition ) ) {
                perror( "Failed to signal about new event while enqueueing event" );
        };

        if( 0 != pthread_mutex_unlock( node->event_mutex ) ) {
                perror( "Failed to unlock event mutex while enqueueing event" );
                return -1;
        };

        return 0;

};
```

4.5.5 Функція отримання події із черги

```
event_t *dequeue_event( node_t *node ) {

        if( 0 == node ) {
                fprintf( stderr, "enqueue_event: node == 0" );
                return 0;
        };

        if( 0 == node->event_mutex ) {
                fprintf( stderr, "enqueue_event: node->event_mutex == 0" );
                return 0;
        };

        if( 0 == node->event_condition ) {
                fprintf( stderr, "enqueue_event: node->event_condition == 0" );
                return 0;
        };

        if( 0 != pthread_mutex_lock( node->event_mutex ) ) {
                perror( "Failed to lock event mutex while dequeueing event" );
                return 0;
        };

        while( 0 == node->event ) {
                pthread_cond_wait( node->event_condition, node->event_mutex );
        };

        event_t *event;

        if( ( node->event == node->event->right ) && ( node->event->right == node->event->left ) ) {

                event       = node->event;
                node->event = 0;

        } else {

                event                           = node->event->left;
                node->event->left->left->right = node->event;
                node->event->left              = node->event->left->left;

        };

        if( 0 != pthread_mutex_unlock( node->event_mutex ) ) {
                perror( "Failed to unlock event mutex while dequeueing event" );
                return 0;
        };

        return event;

};
```

4.6 Реалізація списку з'єднань

4.6.1 Реалізація структури з'єднання

```
typedef struct connection_t {
	int socket;
	int type;
	struct connection_t *left;
	struct connection_t *right;
} connection_t;
```

4.6.2 Функція створення з'єднання

```
connection_t *create_connection( int socket, int type ) {

	if( -1 == socket ) {
		fprintf( stderr, "create_connection: socket == -1\n" );
		return 0;
	};

	connection_t *connection = malloc( sizeof( connection_t ) );

	if( 0 == connection ) {
		perror( "Failed to allocate memory while creating connection" );
		close( socket );
		return 0;
	};

	struct timeval timeout;

	timeout.tv_sec  = NODE_CLIENT_SOCKET_TIMEOUT;
	timeout.tv_usec = 0;

	if( -1 == setsockopt( socket, SOL_SOCKET, SO_RCVTIMEO, ( char * ) &timeout,
sizeof( timeout ) ) ) {
		perror( "Failed to set client socket read timeout" );
	};

	if( -1 == setsockopt( socket, SOL_SOCKET, SO_SNDTIMEO, ( char * ) &timeout,
sizeof( timeout ) ) ) {
		perror( "Failed to set client socket write timeout" );
	};

	connection->socket = socket;
	connection->type   = type;

	connection->left   = 0;
	connection->right  = 0;

	return connection;

};
```

4.6.3 Процедура переривання з'єднання

```
int destroy_connection( node_t *node, int socket ) {

	if( 0 == node ) {
		fprintf( stderr, "destroy_connection: node == 0" );
		return -1;
	};

	if( -1 == socket ) {
		fprintf( stderr, "destroy_connection: socket == -1" );
		return -1;
	};

	if( 0 == node->connection_mutex ) {
		fprintf( stderr, "destroy_connection: node->connection_mutex == 0" );
		return -1;
	};

	if( 0 != pthread_mutex_lock( node->connection_mutex ) ) {
		perror( "Failed to lock connection mutex while destroying connection" );
		return -2;
	};

	int destroyed = -1;

	if( 0 != node->connection ) {

		connection_t *current = node->connection->right;

		do {

			if( current->socket == socket ) {

				close( socket );

				if( current->left == current->right ) {
					free( node->connection );
					node->connection = 0;
				} else {
					current->left->right = current->right;
					current->right->left = current->left;
					free( current );
				};

				destroyed = 0;

				break;

			};

		} while( current != node->connection );

	};

	if( 0 != pthread_mutex_unlock( node->connection_mutex ) ) {
		perror( "Failed to unlock connection mutex while destroying connection" );
		return -2;
	};

	if( -1 == destroy_events( node, socket ) ) {
		return -2;
	};

	return destroyed;

};
```

4.6.4 Функція додавання з'єднання до списку

```
int add_connection( node_t *node, connection_t *connection ) {

        if( 0 == node ) {
                fprintf( stderr, "add_connection: node == 0" );
                return 0;
        };

        if( 0 == connection ) {
                fprintf( stderr, "add_connection: connection == 0" );
                return 0;
        };

        if( 0 == node->connection_mutex ) {
                fprintf( stderr, "add_connection: node->connection_mutex == 0" );
                return 0;
        };

        if( 0 != pthread_mutex_lock( node->connection_mutex ) ) {
                perror( "Failed to lock node connection mutex while adding connection" );
                return -1;
        };

        if( 0 == node->connection ) {
                connection->left  = connection;
                connection->right = connection;
                node->connection  = connection;
        } else {
                connection->left              = node->connection;
                connection->right             = node->connection->right;
                node->connection->right->left = connection;
                node->connection->right       = connection;
        };

        if( 0 != pthread_mutex_unlock( node->connection_mutex ) ) {
                perror( "Failed to lock node connection mutex while adding connection" );
                return -1;
        };

        return 0;

};
```

4.6.5 Функція зміни типу з'єднання

```
int set_connection_type( node_t *node, int socket, int type ) {

	if( 0 == node ) {
		fprintf( stderr, "set_connection_type: node == 0" );
		return 0;
	};

	if( -1 == socket ) {
		fprintf( stderr, "set_connection_type: socket == -1" );
		return 0;
	};

	if( 0 == node->connection_mutex ) {
		fprintf( stderr, "get_connection_type: node->connection_mutex == 0" );
		return 0;
	};

	if( 0 != pthread_mutex_lock( node->connection_mutex ) ) {
		perror( "Failed to lock connection mutex while setting connection type" );
		return -1;
	};

	if( 0 == node->connection ) {
		return 0;
	};

	connection_t *current = node->connection->right;

	do {

		if( current->socket == socket ) {
			current->type = type;
			break;
		};

		current = current->right;

	} while( current != node->connection );

	if( 0 != pthread_mutex_unlock( node->connection_mutex ) ) {
		perror( "Failed to unlock connection mutex while setting connection type" );
		return -1;
	};

	return 0;

};
```

4.6.6 Функція отримання типу з'єднання

```
int get_connection_type( node_t *node, int socket ) {

	if( 0 == node ) {
		fprintf( stderr, "get_connection_type: node == 0" );
		return -1;
	};

	if( -1 == socket ) {
		fprintf( stderr, "get_connection_type: socket == -1" );
		return -1;
	};

	if( 0 == node->connection_mutex ) {
		fprintf( stderr, "get_connection_type: node->connection_mutex == 0" );
		return -1;
	};

	if( 0 != pthread_mutex_lock( node->connection_mutex ) ) {
		perror( "Failed to lock connection mutex while getting connection type" );
		return -2;
	};

	if( 0 == node->connection ) {
		return -1;
	};

	connection_t *current = node->connection->right;

	int type = -1;

	do {

		if( current->socket == socket ) {
			type = current->type;
			break;
		};

		current = current->right;

	} while( current != node->connection );

	if( 0 != pthread_mutex_unlock( node->connection_mutex ) ) {
		perror( "Failed to unlock connection mutex while getting connection type" );
		return -2;
	};

	return type;

};
```

4.7 Процедура авторизації

```
void client_authentification( node_t *node, event_t *event ) {

	if( 0 == node ) {
		fprintf( stderr, "client_authentification: node == 0\n" );
		return;
	};

	if( 0 == event ) {
		fprintf( stderr, "client_authentification: event == 0\n" );
		return;
	};

	unsigned char challenge_length = NODE_CHALLENGE_MIN_LENGTH + rand() %
( NODE_CHALLENGE_MAX_LENGTH - NODE_CHALLENGE_MIN_LENGTH );

	if( -1 == socket_write( node, event->socket, ( char * ) &challenge_length, 1 ) ) {
		return;
	};

	char *challenge = create_challenge( challenge_length );

	if( 0 == challenge ) {
		destroy_connection( node, event->socket );
		return;
	};

	if( -1 == socket_write( node, event->socket, challenge, challenge_length ) ) {
		free( challenge );
		return;
	};

	char *response = socket_read( node, event->socket, NODE_HASH_LENGTH );

	if( 0 == response ) {
		return;
	};

	char buffer[NODE_HASH_LENGTH];

	MHASH hasher;

	hasher = mhash_hmac_init( MHASH_SHA512, node->read_key, node->read_key_length,
mhash_get_hash_pblock( MHASH_SHA512 ) );
	mhash( hasher, challenge, challenge_length );
	mhash_hmac_deinit( hasher, buffer );

	if( compare( response, buffer, NODE_HASH_LENGTH ) ) {

		free( challenge );
		free( response );

		if( -1 == authentification_finish( node, event->socket, node->read_key,
node->read_key_length ) ) {
			return;
		};

		if( -1 == set_connection_type( node, event->socket, NODE_CONNECTION_READONLY ) ) {
			destroy_connection( node, event->socket );
			return;
		};

		return;

	};

	hasher = mhash_hmac_init( MHASH_SHA512, node->write_key, node->write_key_length,
mhash_get_hash_pblock( MHASH_SHA512 ) );
	mhash( hasher, challenge, challenge_length );
	mhash_hmac_deinit( hasher, buffer );
```

```
        if( compare( response, buffer, NODE_HASH_LENGTH ) ) {

                free( challenge );
                free( response );

                if( -1 == authentification_finish( node, event->socket, node->write_key,
node->write_key_length ) ) {
                        return;
                };

                if( -1 == set_connection_type( node, event->socket, NODE_CONNECTION_READWRITE ) ) {
                        destroy_connection( node, event->socket );
                        return;
                };

        };
        hasher = mhash_hmac_init( MHASH_SHA512, node->node_key, node->node_key_length,
mhash_get_hash_pblock( MHASH_SHA512 ) );
        mhash( hasher, challenge, challenge_length );
        mhash_hmac_deinit( hasher, buffer );

        if( compare( response, buffer, NODE_HASH_LENGTH ) ) {

                free( challenge );
                free( response );

                if( -1 == authentification_finish( node, event->socket, node->node_key,
node->node_key_length ) ) {
                        return;
                };

                if( -1 == set_connection_type( node, event->socket, NODE_CONNECTION_READWRITE ) ) {
                        destroy_connection( node, event->socket );
                        return;
                };

                dispatch_read_command( node, event->socket );
                return;

        };

        destroy_connection( node, event->socket );

};
```

```
int authentification_finish( node_t *node, int socket, char *key, unsigned char key_length ) {

        if( 0 == node ) {
                fprintf( stderr, "authentification_finish: node == 0\n" );
                return 0;
        };

        if( -1 == socket ) {
                fprintf( stderr, "authentification_finish: socket == -1\n" );
                return 0;
        };

        if( 0 == key ) {
                fprintf( stderr, "authentification_finish: key == 0\n" );
                return 0;
        };

        if( 0 == key_length ) {
                fprintf( stderr, "authentification_finish: key_length == 0\n" );
                return 0;
        };

        char *buffer = socket_read( node, socket, 1 );

        if( 0 == buffer ) {
                destroy_connection( node, socket );
                return 0;
        };

        unsigned char challenge_length = ( unsigned char ) buffer[0];

        free( buffer );

        if( 0 == challenge_length ) {
                return -1;
        };

        char *challenge = socket_read( node, socket, challenge_length);

        if( 0 == challenge ) {
                return -1;
        };

        MHASH hasher;

        char hash[NODE_HASH_LENGTH];

        hasher = mhash_hmac_init( MHASH_SHA512, key, key_length, mhash_get_hash_pblock( MHASH_SHA512 )
);
        mhash( hasher, challenge, challenge_length );
        mhash_hmac_deinit( hasher, hash );

        if( -1 == socket_write( node, socket, hash, NODE_HASH_LENGTH ) ) {
                free( challenge );
                return -1;
        };

        return 0;

};
```

4.8 Процедура видачі даних

```
void client_get_chunk( node_t *node, event_t *event ) {

	if( 0 == node ) {
		fprintf( stderr, "client_get_chunk: node == 0\n" );
		return;
	};

	if( 0 == event ) {
		fprintf( stderr, "client_get_chunk: event == 0\n" );
		return;
	};

	if( NODE_CONNECTION_READONLY > get_connection_type( node, event->socket ) ) {
		destroy_connection( node, event->socket );
		return;
	};

	char *chunk_hash = socket_read( node, event->socket, NODE_HASH_LENGTH );

	if( 0 == chunk_hash ) {
		return;
	};

	char *chunk_hash_hex = unpack_hash( chunk_hash );

	char *chunk_path = strcat( NODE_DATA_PATH, chunk_hash_hex );

	struct stat chunk_stat;

	if( -1 == stat( chunk_path, &chunk_stat ) ) {
		char response = 0;
		socket_write( node, event->socket, &response, 1 );
		// proxy to neighbours
		return;
	};

	FILE *chunk_file = fopen( chunk_path, "r" );

	if( 0 == chunk_file ) {
		perror( "Failed to open chunk file for reading" );
		destroy_connection( node, event->socket );
		return;
	};

	char *chunk_contents = malloc( chunk_stat.st_size );

	if( 0 == chunk_contents ) {
		perror( "Failed to allocate memory for chunk contents" );
		fclose( chunk_file );
		destroy_connection( node, event->socket );
		return;
	};

	if( 1 != fread( chunk_contents, chunk_stat.st_size, 1, chunk_file ) ) {
		free( chunk_contents );
		fclose( chunk_file );
		destroy_connection( node, event->socket );
		return;
	};

	fclose( chunk_file );

	unsigned int chunk_given = 0;
```

```
        while( chunk_stat.st_size ) {

                unsigned char part_size = chunk_stat.st_size > 255 ? 255 : chunk_stat.st_size;

                if( -1 == socket_write( node, event->socket, ( char * ) &part_size, 1 ) ) {
                        free( chunk_contents );
                        return;
                };

                if( -1 == socket_write( node, event->socket, &chunk_contents[chunk_given], part_size )
) {
                        free( chunk_contents );
                        return;
                };

                chunk_given += part_size;

        };

        dispatch_read_command( node, event->socket );

};
```

4.9 Процедура збереження даних

```
void client_put_chunk( node_t *node, event_t *event ) {

	if( 0 == node ) {
		fprintf( stderr, "client_put_chunk: node == 0\n" );
		return;
	};

	if( 0 == event ) {
		fprintf( stderr, "client_put_chunk: event == 0\n" );
		return;
	};

	if( NODE_CONNECTION_READWRITE > get_connection_type( node, event->socket ) ) {
		destroy_connection( node, event->socket );
		return;
	};

	char *chunk_hash = socket_read( node, event->socket, NODE_HASH_LENGTH );

	if( 0 == chunk_hash ) {
		return;
	};

	char *chunk_hash_hex = unpack_hash( chunk_hash );

	if( 0 == chunk_hash_hex ) {
		destroy_connection( node, event->socket );
		return;
	};

	char *chunk_path = strcat( NODE_DATA_PATH, chunk_hash_hex );

	struct stat chunk_stat;

	char chunk_exists = 0;

	if( -1 != stat( chunk_path, &chunk_stat ) ) {
		socket_write( node, event->socket, &chunk_exists, 1 );
		return;
	} else {
		if( -1 == socket_write( node, event->socket, &chunk_exists, 1 ) ) {
			return;
		};
	};

	char *chunk_contents = 0;
	unsigned int chunk_size = 0;
	unsigned char part_size = 0;

	for( ; ; ) {

		char *buffer = socket_read( node, event->socket, 1 );

		if( 0 == buffer ) {
			if( 0 != chunk_contents ) {
				free( chunk_contents );
			};
			return;
		};

		part_size = buffer[0];

		if( 0 == part_size ) {
			break;
		};

		chunk_contents = realloc( chunk_contents, chunk_size + part_size );

		free( buffer );

		buffer = socket_read( node, event->socket, part_size );
```

```
                if( 0 == buffer ) {
                        if( 0 != chunk_contents ) {
                                free( chunk_contents );
                        };
                        return;
                };

                memcpy( &chunk_contents[chunk_size], buffer, part_size );

                free( buffer );

                chunk_size += part_size;

        };

        char real_chunk_hash[NODE_HASH_LENGTH];

        MHASH hasher = mhash_init( MHASH_SHA512 );
        mhash( hasher, chunk_contents, chunk_size );
        mhash_deinit( hasher, real_chunk_hash );

        char status = 0;

        if( ! compare( real_chunk_hash, chunk_hash, NODE_HASH_LENGTH ) ) {
                socket_write( node, event->socket, &status, 1 );
                dispatch_read_command( node, event->socket );
                return;
        };

        FILE *chunk_file = fopen( chunk_path, "w" );

        if( 1 != fwrite( chunk_contents, chunk_size, 1, chunk_file ) ) {
                socket_write( node, event->socket, &status, 1 );
                dispatch_read_command( node, event->socket );
                return;
        };

        if( -1 == fclose( chunk_file ) ) {
                socket_write( node, event->socket, &status, 1 );
                dispatch_read_command( node, event->socket );
                return;
        };

        status = 1;

        socket_write( node, event->socket, &status, 1 );

        dispatch_read_command( node, event->socket );

};
```

4.10 Процедура видалення даних

```
void client_drop_chunk( node_t *node, event_t *event ) {

	if( 0 == node ) {
		fprintf( stderr, "client_drop_chunk: node == 0\n" );
		return;
	};

	if( 0 == event ) {
		fprintf( stderr, "client_drop_chunk: event == 0\n" );
		return;
	};

	if( NODE_CONNECTION_READWRITE > get_connection_type( node, event->socket ) ) {
		destroy_connection( node, event->socket );
		return;
	};

	char *chunk_hash = socket_read( node, event->socket, NODE_HASH_LENGTH );

	if( 0 == chunk_hash ) {
		return;
	};

	char *chunk_hash_hex = unpack_hash( chunk_hash );

	char *chunk_path = strcat( NODE_DATA_PATH, chunk_hash_hex );

	unlink( chunk_path );

	dispatch_read_command( node, event->socket );

};
```

4.11 Допоміжні процедури та функції

```
unsigned char compare( char *this, char *that, unsigned int length ) {
	unsigned int i = 0;
	for( ; i < length; i++ ) {
		if( this[i] ^ that[i] ) {
			return 0;
		};
	};
	return 1;
};

char *socket_read( node_t *node, int socket, unsigned char length ) {

	if( 0 == node ) {
		fprintf( stderr, "socket_read: node == 0\n" );
		return 0;
	};

	if( -1 == socket ) {
		fprintf( stderr, "socket_read: socket == -1\n" );
		return 0;
	};

	if( 0 == length ) {
		fprintf( stderr, "socket_read: length == 0\n" );
		return 0;
	};

	char *data = malloc( length );

	if( 0 == data ) {
		perror( "Failed to allocate memory for buffer while reading from socket" );
		return 0;
	};

	if( length != recv( socket, data, length, 0 ) ) {
		free( data );
		destroy_connection( node, socket );
		return 0;
	};

	return data;

};
```

```
int socket_write( node_t *node, int socket, char *data, unsigned char length ) {

	if( 0 == node ) {
		fprintf( stderr, "socket_write: node == 0\n" );
		return 0;
	};

	if( -1 == socket ) {
		fprintf( stderr, "socket_write: socket == -1\n" );
		return 0;
	};

	if( 0 == data ) {
		fprintf( stderr, "socket_write: data == 0\n" );
		return 0;
	};

	if( 0 == length ) {
		fprintf( stderr, "socket_write: length == 0\n" );
		return 0;
	};

	if( length != send( socket, data, length, 0 ) ) {
		destroy_connection( node, socket );
		return -1;
	};

	return 0;

};

char *create_challenge( unsigned char length ) {

	if( 0 == length ) {
		fprintf( stderr, "create_challenge: length == 0\n" );
		return 0;
	};

	char *challenge = malloc( length );

	if( 0 == challenge ) {
		perror( "Failed to allocate memory for challenge" );
		return 0;
	};

	unsigned char i = 0;

	for( ; i < length; i++ ) {
		challenge[i] = rand() % 255;
	};

	return challenge;

};

void hex_dump( char *data, unsigned int length ) {
	unsigned int i = 0;
	for( ; i < length; i++ ) {
		printf( "%02X ", ( unsigned char ) data[i] );
	};
	printf( "\n" );
};
```

```
void dispatch_read_command( node_t *node, int socket ) {

	if( 0 == node ) {
		fprintf( stderr, "dispatch_read_command: node == 0\n" );
		return;
	};

	if( -1 == socket ) {
		fprintf( stderr, "dispatch_read_command: socket == -1\n" );
		return;
	};

	event_t *event = create_event( socket, 5 );

	if( 0 != event ) {
		enqueue_event( node, event );
	};

};

char *unpack_hash( char *hash ) {

	if( 0 == hash ) {
		fprintf( stderr, "unpack_hash: hash == 0\n" );
		return 0;
	};

	char *hash_hex = malloc( NODE_HASH_LENGTH * 2 + 1 );

	if( 0 == hash_hex ) {
		perror( "Failed to allocate memory for unpacked hash" );
		return 0;
	};

	hash_hex[ NODE_HASH_LENGTH * 2 ] = 0;

	unsigned int i = 0;

	for( ; i < NODE_HASH_LENGTH; i++ ) {
		sprintf( &hash_hex[ i * 2 ], "%02X", ( unsigned char ) hash[i] );
	};

	return hash_hex;

};

void dispatch_get_chunk( node_t *node, int socket ) {

	if( 0 == node ) {
		fprintf( stderr, "dispatch_get_chunk: node == 0\n" );
		return;
	};

	if( -1 == socket ) {
		fprintf( stderr, "dispatch_get_chunk: socket == -1\n" );
		return;
	};

	event_t *event = create_event( socket, 2 );

	if( 0 != event ) {
		enqueue_event( node, event );
	};

};
```

```
void dispatch_put_chunk( node_t *node, int socket ) {

	if( 0 == node ) {
		fprintf( stderr, "dispatch_put_chunk: node == 0\n" );
		return;
	};

	if( -1 == socket ) {
		fprintf( stderr, "dispatch_put_chunk: socket == -1\n" );
		return;
	};

	event_t *event = create_event( socket, 3 );

	if( 0 != event ) {
		enqueue_event( node, event );
	};

};

void dispatch_drop_chunk( node_t *node, int socket ) {

	if( 0 == node ) {
		fprintf( stderr, "dispatch_drop_chunk: node == 0\n" );
		return;
	};

	if( -1 == socket ) {
		fprintf( stderr, "dispatch_drop_chunk: socket == -1\n" );
		return;
	};

	event_t *event = create_event( socket, 4 );

	if( 0 != event ) {
		enqueue_event( node, event );
	};

};

void client_read_command( node_t *node, event_t *event ){

	if( 0 == node ) {
		fprintf( stderr, "client_read_command: node == 0\n" );
		return;
	};

	if( 0 == event ) {
		fprintf( stderr, "client_read_command: event == 0\n" );
		return;
	};

	char command[1];

	if( 1 == recv( event->socket, command, 1, 0 ) ) {

		switch( command[0] ) {
			case 0: dispatch_get_chunk( node, event->socket ); break; // get chunk
			case 1: dispatch_put_chunk( node, event->socket ); break; // put chunk
			case 2: dispatch_drop_chunk( node, event->socket ); break; // drop chunk
			default: destroy_connection( node, event->socket );
		};

	};

};
```

5 ОХОРОНА ПРАЦІ

5.1 Загальна характеристика умов праці програміста / системного адміністратора

Науково-технічний прогрес вніс серйозні зміни в умови виробничої діяльності працівників розумової праці. Їх праця стала більш інтенсивною, напруженою, вона вимагає значних витрат розумової та емоційної енергії. Це потребує комплексного рішення проблем ергономіки, гігієни і організації праці, регламентації режимів праці і відпочинку.

Робота з комп'ютером характеризується значною розумовою напругою і нервово-емоційним навантаженням операторів (програмістів, системних адміністраторів), високою напруженістю зорової роботи і достатньо великим навантаженням на м'язи рук при роботі з клавіатурою ЕОМ. Велике значення має раціональна конструкція і розташування елементів робочого місця, що важливе для підтримки оптимальної робочої пози оператора.

В процесі роботи з комп'ютером необхідно дотримуватися правильного режиму праці та відпочинку. Інакше у персоналу можуть виникнути скарги на незадоволеність роботою, головні болі, дратівливість, порушення сну, утомленість і хворобливі відчуття в очах, в попереку, в області шиї та руках.

5.2 Вимоги до виробничих приміщень

5.2.1 Кольори та коефіцієнт віддзеркалення

Колір приміщень і меблів повинен сприяти створенню сприятливих умов для зорового сприйняття та гарного настрою.

Джерела світла, які дають віддзеркалення від поверхні екрану, значно погіршують точність сприйняття знаків на екрані монітору чи на клавіатурі та спричиняють перешкоди фізіологічного характеру, які можуть проявитися в значній напрузі, особливо при тривалій роботі. Віддзеркалення, включаючи віддзеркалення від вторинних джерел світла, повинне бути зведено до мінімуму. Для захисту від надмірної яскравості вікон можуть бути застосовані штори або жалюзі.

В приміщеннях, де знаходиться комп'ютер, необхідно забезпечити наступний коефіцієнт віддзеркалення:

- стеля — 60-70%
- стіни — 40-50%
- підлога — приблизно 30%
- інші поверхні, меблі — 30-40%

5.2.2 Освітлення

Згідно із ДБН В.2.5-28-2006, при виконанні робіт категорії високої зорової точності (найменший розмір об'єкту розрізнення 0,3-0,5мм) величина коефіцієнта природного освітлення (КЕО) повинна бути не нижчою 1,2%, а при зоровій роботі середньої точності (найменший розмір об'єкту розрізнення 0,5-1,0 мм) КЕО повинен бути не нижчим 0,9%.

Згідно пункту 4.7 ДБН В.2.5-28-2006, для робочого місця оператора ЕОМ необхідно застосовувати комбіноване освітлення.

У якості джерела штучного освітлення звичайно використовуються люмінесцентні лампи типа ЛБ, які попарно об'єднуються в світильники, які повинні розташовуватися рівномірно над робочими поверхнями.

Вимоги до освітленості в приміщеннях, де встановлені комп'ютери, наступні: на робочих столах — 400 лк, на екрані монітору — 200 лк.

Ступінь освітлення приміщення і яскравість екрану комп'ютера повинні бути приблизно однаковими, оскільки яскраве світло в районі периферійного зору значно збільшує напруженість очей і, як наслідок, приводить до їх швидкої стомлюваності.

5.2.3 Мікроклімат

Обчислювальна техніка є джерелом істотних тепловиділень, що може привести до підвищення температури і зниження відносної вологості в приміщенні. В приміщеннях, де встановлені комп'ютери, повинні дотримуватися певні параметри мікроклімату. В санітарних нормах встановлені величини

параметрів мікроклімату, що створюють комфортні умови. Ці норми встановлюються залежно від пори року, характеру трудового процесу і характеру виробничого приміщення (згідно ДСН 3.3.6.042-99 — оптимальні величини показників мікроклімату):

Таблиця 1 — Оптимальні величини показників мікроклімату

Пора року	Категорія робіт	Температура повітря, °С	Відносна вологість повітря, %	Швидкість руху повітря, м/с
Холодна	легка-1а	22-24	40-60	0,1
	легка-1б	21-23	40-60	0,1
Тепла	легка-1а	23-25	40-60	0,1
	легка-1б	22-24	40-60	0,2

Згідно пункту 1.1.4 ДСН 3.3.6.042-99, при виконанні робіт операторського типу повинні дотримуватися оптимальні умови мікроклімату: температура повітря 22-24°С, відносна вологість 60-40%, швидкість руху повітря не більш 0,1 м/сек.

Згідно ДСанПІН 3.3.2.007-98, п. 2.3, об'єм приміщень, в яких працюють оператори, повинен бути не меншим за 20 м3/людину з урахуванням максимального числа одночасно працюючих в зміну. Площа на одного робітника — не менше 6 м2.

5.2.4 Шум і вібрація

Згідно ГОСТ 12.1.003-83 рівень шуму на робочому місці програмістів / системних адміністраторів не повинен перевищувати 50 дБА.

Для зниження рівня шуму, стіни і стеля приміщень, де встановлені комп'ютери, можуть бути облицьовані звукопоглинаючими матеріалами, якщо робочі приміщення операторів ЕОМ межують із шумними виробничими

приміщеннями (наприклад, серверними залами центрів обробки даних).

Джерела вібрації на робочому місці оператора ЕОМ відсутні.

5.3 Ергономічні вимоги до робочого місця

Створення комфортних умов для роботи програміста / системного адміністратора є надзвичайно важливим, так як фізичний дискомфорт приводить до негативної емоційної реакції, що відчутно впливає на продуктивність роботи. Також, незручна робоча поза може з часом призвести до виникнення різних захворювань (викривлення хребту, захворювання суставів, особливо кисті, захворювання зорового апарату тощо).

Гігієнічні вимоги до робочого місця операторів ЕОМ регламентуються пунктом 4 ДСанПІН 3.3.2.007-98.

Робоче місце має забезпечити підтримання оптимальної робочої пози. Воно має бути розташоване так, щоб природне світло падало збоку (переважно зліва). Робоче місце не може бути розташоване так, щоб робітник був розміщений спиною до джерела світла: це приводить до утворення відблисків на екрані монітору, що негативно впливає на сприйняття інформації людиною та її зоровий апарат.

Природний світловий потік, має бути регульованим: зовнішні козирки на вікнах, штори, жалюзі.

Мінімальні відстані між моніторами на робочих місцях рекомендується дотримувати такі: 1,2 м між їх бічними поверхнями та 2,5 м між їх тильними сторонами. Робітники повинні мати можливість вільно пересуватися по виробничому приміщенню (між робочими місцями), на одного робітника має припадати 6 м2 площі.

Робоча поверхня має бути розташована на відстані мінімум 680 мм (бажане регулювання в межах 680...800 мм). Рекомендовані розміри робочої поверхні — 600...1400 х 800...1000 мм.

Робочий стіл повинен забезпечувати простір для ніг із висотою від 600 мм, шириною від 50 мм, глибиною на рівні колін — 450 мм, глибиною на рівні простягнутої ноги — 650 мм.

Серед комп'ютерних столів можна порекомендувати наступні:

ErgoLife 501-11 B,S 156

Рисунок 13 — Комп'ютерний стіл ErgoLife 501-11 B,S 156

Виступ для підтримання лівої руки
Регульований по висоті
Ширина: 1560 мм
Висота: 630 - 1200 мм
Глибина: 730 мм
Під'омна вага: 150 кг

ErgoLife 501-11 B,S 156-156

Рисунок 14 — Комп'ютерний стіл ErgoLife 501-11 B,S 156-156

Аналогічний моделі, описаній зліва, відмінності:
виступ для підтримки правої руки
надійніша підставка (три точки опори)

Для зберігання документів, поряд зі столом необхідно розташувати невелику тумбочку. Розташування тумбочки не повинно перешкоджати переміщенню робітників між робочими місцями.

Робочий стілець/крісло має бути під'йомно-поворотним, регульованим по висоті, із регульованими нахилами сидіння та спинки. Регулювання кожного параметру має бути легким та надійно фіксуватися. Лінійні параметри повинні мати крок регулювання 15...20 мм, кутові — 2...5 градусів. Передній край сидіння має бути заокругленим.

Висота поверхні сидіння має регулюватися в межах 400...500 мм, а ширина і глибина становити не менше, ніж 400 мм. Кут нахилу сидіння має регулюватися до 15 градусів вперед і до 5 градусів назад.

Висота спинки стільця має становити 300±20 мм, ширина — не менше, ніж 380 мм, радіус кривизни горизонтальної площини — 400 мм. Кут нахилу спинки має регулюватися в межах 1...30 градусів від вертикального положення. Відстань від спинки до переднього краю сидіння має регулюватися в межах 260...400 мм.

Для зменшення статичного напруження рук оператора ЕОМ, слід використовувати стаціонарні або змінні підлокітники завдовжки не менше ніж 250 мм, завширшки 50...70 мм, що регулюються за висотою над сидінням у межах 230...260 мм і відстанню між підлокітниками в межах 350...500 мм.

Поверхня сидіння і спинки стільця має бути напівм'якою з нековзним, повітронепроникним покриттям, що легко чиститься і не електризується.

Серед комп'ютерних стільців можна можна рекомендувати наступні моделі:

BodyBilt J2407

Рисунок 15 — Комп'ютерний стілець BodyBilt J2407

- Регульована висота поверхні стільця
- Регульований нахил спинки
- Регульована відстань від поверхні стільця до спинки
- Регульована висота підлокітників
- Матеріал поверхонь — тканина

BodyBilt K3507

Рисунок 16 — Комп'ютерний стілець BodyBilt K3507

- Аналогічно J2407, але із підголівником
- Регульований виступ підголівника відносно спинки
- Регульована висота підголівника відносно спинки
- Матеріал — штучна шкіра (легше чистити)

Робоче місце має бути обладнане підставкою для ніг завширшки не менше ніж 300 мм, завглибшки не менше ніж 400 мм, що регулюється за висотою в межах до 150 мм і за кутом нахилу опорної поверхні підставки до 20 градусів. Підставка повинна мати рифлену поверхню і бортик по передньому краю заввишки 10 мм.

Екран монітору має розташовуватися на оптимальній відстані від очей оператора, що становить 600...700 мм, але не ближче, ніж 600 мм з урахуванням розміру літерно-цифрових знаків і символів.

Монітор має бути розташованим так, щоб була забезпечена зручність сприйняття інформації із кутом віхилення до +30 градусів у вертикальній площині відносно нормальної лінії погляду працівника. Регульована позиція (нахил) монітору є дуже бажаним.

Розповсюджені монітори із співвідношеннями сторін 4:3 та 16:9. Перше співвідношення найбільш підходить для редагування одного документу (розвернутого на повний екран). Друге співвідношення (16:9) підходить для одночасного перегляду 2 документів на одному моніторі. Для програміста або системного адміністратора більш бажаним є монітор із співвідношенням сторін 16:9, так як, їм здебільшого бажано мати відкритим редактор програмного коду або командний рядок поряд із документацією або програмою моніторингу сервера або мережі. Це допомагає тримати у полі зору необхідні програми та зручно і швидко перемикати увагу між ними.

Для комфортного сприймання текстової інформації бажаний монітор із великим значенням DPI: це дозволяє досягти гладкості шрифту, що сприймається приємніше, ніж напівквадратні шрифти.

Монітор має відображати велику кількість кольорів: це дозволяє ефективно використовувати згладжування шрифтів, що позитивно впливає на сприймання текстової інформації — це дещо знижує контрастність тексту із його фоном (в адекватних межах), що менше навантажує зоровий апарат людини.

Підсвітка екрану монітору має бути білою (для правильної передачі кольорів) та рівномірною по всій поверхні екрану.

Серед останніх моделей моніторів для програмістів та системних

адміністраторів можна рекомендувати, наприклад, такі:

Samsung B2430L

Рисунок 17 — Монітор Samsung B2430L

Пропорції екрану: 16:9
Діагональ: 24 дюйми
Роздільна здатність: 1080p
Кількість кольорів: 16 млн.
Регульований кут нахилу
Монітор є зручним для одночасного запуску двох програм завдяки співвідношенню сторін. Зручний для програмістів/системних адміністраторів — дозволяє одночасно переглядати код програми та, наприклад, довідку.

LG E1910S

Рисунок 18 — Монітор LG E1910S

Пропорції екрану: 5:4
Діагональ: 18.9 дюймів
Роздільна здатність: 1280x1024
Кількість кольорів: 16 млн.
Регульований кут нахилу
Монітор є зручним для редагування текстових документів, перегляд лише одного документу на екрані. Є зручним для операторів комп'ютерного набору.

Клавіатуру слід розташовувати на поверхні столу на відстані 100...300 мм від краю, звернутого до оператора. У конструкції клавіатури має передбачатися опорний пристрій (виготовлений із матеріалу з високим коефіцієнтом тертя, що перешкоджає випадковому її зсуву), який дає змогу змінювати кут нахилу поверхні клавіатури у межах 5...15 градусів. Висота середнього рядка клавіш має не перевищувати 30 мм. Поверхня клавіатури має бути матовою з коефіцієнтом відбиття 0,4.

Надписи на клавішах клавіатури мають бути розбірливими, добре контрастувати із кольором клавіш.

Для роботи можна використовувати спеціальні ергономічні клавіатури, групи клавіш (ліва та права група) на яких розташовані під деяким кутом відносно центру клавіатури — відповідно до куту рук оператора (лікті дещо

розведені в сторони). Це дозволяє зменшити кількість необхідних рухів кисті у горизонтальній поверхні (вліво-вправо) для охоплення усього діапазону клавіш.

Крім того, іншою особливістю ергономічних клавіатур є вигини у вертикальній площині — наприклад, вертикальний підйом по центру, що позитивно впливає на руки оператора.

Також є спеціальні ергономічні клавіатури, в яких ліва і права групи клавіш рознесені аж до декількох сантиметрів. Хоча, найзручнішими клавіатурами вважаються симетричні клавіатури, із невеликим підйомом по центру, із недалеко рознесеними групами клавіш, при чому групи клавіш мають бути дещо нахилені відповідно до позиції рук оператора.

Серед моделей клавіатур можна порекомендувати, наприклад, такі:

Adesso Intellimedia Pro Ergo Keyboard

Рисунок 19 — Комп'ютерна клавіатура Adesso Intellimedia Pro Ergo Keyboard

Microsoft Natural Keyboard Elite White A11-00337

Рисунок 20 — Комп'ютерна клавіатура Microsoft Natural Keyboard Elite White A11-00337

Окремий цифровий блок	Аналогічна клавіатурі Adesso, відмінності:
Вигин клавіатури під кут розташування рук оператора	симетричне розташування правого і лівого блоків клавіш
Підвищення в центральній частині	
Мультимедійні клавіші (надлишковий функціонал для програміста/системного адміністратора)	немає зайвих мультимедійних клавіш

Іншим важливим компонентом робочого обладнання програміста / системного адміністратора є комп'ютерна мишка, до вибору якої слід підійти

серйозно, так як незручна мишка (як і клавіатура) може призвести до захворювань кисті (тунельного синдрому тощо).

Найбільш помітний фактор при виборі миші — це її розмір. Його необхідно підбирати відповідно до розміру руки оператора.

Миша повинна бути достатньо довгою, щоб пальці оператора не були постійно зігнуті і достатньо широкою, щоб не кисть оператора не дуже вигиналася у спробі охопити корпус мишки.

Хід клавіш та коліщатка миші повинен бути легким, але не занадто (для запобігання реакції ЕОМ при випадковому доторку до клавіш або коліщатка). Бажано, щоб натиснення клавіш супроводжуватися негучним звуком (кліком), що дозволяє оператору відчути відгук маніпулятора (що клавіша була дійсно натиснута).

Спеціальні ергономічні миші часто роблять несиметричними. Крім того, деякі виробники пропонують дуже незвичні форми для цього маніпулятора.

Серед моделей комп'ютерних мишок варто звернути увагу на такі моделі:

Logitech Performance Mouse MX

Рисунок 21 — Комп'ютерна миша Logitech Performance Mouse MX

Асиметричний корпус
Підставка для великого пальця оператора
Додаткові клавіші для скроллінгу

Microsoft Wireless Laser 6000 v2

Рисунок 22 — Комп'ютерна миша Microsoft Wireless Laser 6000 v2

Асиметричний корпус
Підставка для великого пальця
Заглиблення для зручної фіксації мізинця та безіменного пальців

Коврик для мишки повинен не утруднювати рух миші по його поверхні, має бути таким, щоб мишка правильно визначала рух (курсор на екрані пересувався відповідно до її руху). Коврик не повинен ковзати по робочій поверхні при переміщенні миші.

Взаємне розташування клавіатури та миші рекомендується робити якомога меншим — чим ближче мишка до клавіатури, там краще для рук оператора. Але не варт розміщувати їх занадто близько одна до одної для запобігання натисненням при випадкових рухах.

На робочій поверхні оператора не повинні знаходитися предмети, які перешкоджають сприйняття інформації із екрану монітору. Робоче місце програміста / системного адміністратора рекомендовано обладнати пюпітром для документів, що легко переміщуються.

Загальні вимоги до кольору комп'ютерного обладнання, яке знаходиться в полі зору оператора ЕОМ — він має бути спокійним, м'яким. Обладнання має розсіювати світло — в ньому не повинно бути деталей, що можуть створювати відблиски.

В зоні видимості оператора не повинно бути предметів, що сильно відволікають увагу — після відволікання потрібен деякий час, щоб знову сфокусуватися на предметі роботи. В той же час, дуже бажано біля монітору розміщувати яскраві предмети теплого кольору (наприклад, жовті листочки тощо): вони не повинні занадто відволікати від роботи, але час від часу оператор буде кидати на них взір, даючи тим самим відпочинок очам. Окрім того, теплі кольори позитивно впливають на настрій робітника.

При оснащеності робочого місця лазерним принтером, параметри лазерного випромінювання повинні відповідати вимогам СанПіН N 5804-91.

ВИСНОВКИ

При виконанні даної дипломної роботи було розроблено фрагменти програмного забезпечення вузла розподіленої Cloud-мережі, що легко масштабується горизонтально (при додаванні нового вузла у мережу, просто необхідно вказати йому IP-адресу вже підключеного до мережі вузла), для підтримки якої необхідне програмне забезпечення із відкритим кодом (безкоштовне програмне забезпечення), а також яка приховує маршрути руху даних по мережі від користувача (Cloud-система).

Алгоритми, розроблені для програмного забезпеченні такої мережі, були розроблені із оглядом на мову програмування С. Використання компільованої мови програмування дозволяє досягти максимальної швидкості виконання програм у порівнянні, наприклад, із Java.

Окрім того, С, у порівнянні із, наприклад, С++, дозволяє зекономити оперативну пам’ять, необхідну для виконання програми, що досягається за рахунок відсутності високорівневих структур даних (об’єктів), які зазвичай приводять до появи у програмі надлишкових даних, некритичних для її функціонування.

Економія пам’яті також гарно впливає на обслуговування клієнтських з’єднань: частина пам’яті, яка б могла б бути використана програмним забезпеченням вузла, може бути віддана ще одному клієнту.

Тож, результуюча архітектура програмного забезпечення, а також фрагменти підпрограм для виконання певних дій у системі орієнтовані на швидкодію та невеликі вимоги до ресурсів апаратного забезпечення.

У порівнянні із існуючими програмними рішеннями для створення Cloud-мереж, що є універсальними інструментами, призначеними для задоволення якнайширшого кола споживачів, розроблена система вирізняється відсутністю зайвих функцій, а також більшою швидкістю та меншими вимогами до оперативної пам’яті ЕОМ для її запуску.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Клейнрок Л. Теория массового обслуживания, пер. И. И. Глушко / Клейнрок Л. - М.: Машиностроение, 1979 — 432 с.
2. Ивченко Г. И., Каштанов В. А., Коваленко И. Н. Теория массового обслуживания: Учеб. пособие для вузов / Ивченко Г. И., Каштанов В. А., Коваленко И. Н. - М.: Высш. школа, 1982 — 256 с.
3. Бочаров П. П., Печинкин А. В. Теория массового обслуживания: Учебник / Бочаров П. П., Печинкин А. В. - М.: Изд-во РУДН, 1995 — 529 с.
4. Э. Таненбаум, М. ван Стеен. Распределенные системы. Принципы и парадигмы / Э. Таненбаум, М. ван Стеен. - ИД “Питер”, 2003 — 880 с.
5. Керниган Б., Ритчи Д. Язык программирования Си / Керниган Б., Ритчи Д. - Изд-во “Невский диалект”, 2001 — 352 с.
6. Герберт Шилдт. Полный справочник по C / Герберт Шилдт — Изд-во “Вильямс”, 2004 — 704 с.
7. Шон Уолтон. Создание сетевых приложений в среде Linux. Руководство разработчика / Шон Уолтон — Изд-во Вильямс, 2001 — 464 с.
8. У. Р. Стивенс, Б. Феннер, Э. М. Рудофф. UNIX. Разработка сетевых приложений / У. Р. Стивенс, Б. Феннер, Э. М. Рудофф. - ИД “Питер”, 2007 — 1040 с.